VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2019 Tomáš Kovačik

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tomáš Kovačik**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Tieto Czech s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Seidl, Ph.D.**

Konzultant bakalářské práce: Mgr. Zdeněk Dřizga

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 24. dubna 2019 ..... Kosařík .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

Tieto Czech s.r.o.
28. října 3346/91
702 00 Ostrava - Moravská Ostrava
IČO 64608051 DIČ CZ64608051

V Ostravě 16. dubna 2019

.....

Rád bych na tomto místě poděkoval společnosti Tieto Czech s.r.o. a Mgr. Zdeňku Dřizgovi za umožnění absolvování individuální odborné praxe.

V neposlední řadě bych také chtěl poděkovat svému vedoucímu práce Ing. Davidu Seidlovi, Ph.D. za vedení a věcné připomínky k vypracování práce.

Abstrakt

Tato Bakalářská práce popisuje mé působení na individuální odborné praxi ve společnosti Tieto Czech s.r.o., ve které jsem pracoval na pozici Software Developer. Součástí práce je stručný popis této společnosti, mého pracovního zařazení a především projektu, na němž jsem se během své praxe podílel. V rámci něj se v práci následně věnuji popisu jeho prvotní přípravy, výběru technologií a podrobnému popisu zadaných úkolů spolu s řešením, týkající se jak návrhu, tak i implementace jeho částí. Mimo to se zde také zabývám i popisem nasazení celého projektu do produkčního prostředí, a to včetně jeho přípravy a konfigurace. V závěru práce dále shrnuji uplatněné znalosti ze studia na vysoké škole, průběh praxe a dosažené výsledky.

Klíčová slova: webová aplikace, Java, Spring Boot, Spring Data, Elasticsearch, MongoDB, Hibernate, JPA, AWS, Tieto, cloud

Abstract

This Bachelor thesis describes my activities on individual professional practice in the company Tieto Czech s.r.o., in which I worked at Software Developer position. This thesis includes a brief description of this company within my work position and mainly a description of the project on which I was working on during practice. Regarding this, the thesis also includes a description of the first steps related to the project preparation, description of chosen technologies and detailed description of my tasks within solutions, which are related to design as well as an implementation of project parts. Beyond that, I am also focusing here on a description of the project deployment into the production environment within environment preparation and configuration. In conclusion of this thesis, I summarize applied skills from my studies at the university, overall practice and achieved results.

Key Words: web application, Java, Spring Boot, Spring Data, Elasticsearch, MongoDB, Hibernate, JPA, AWS, Tieto, cloud

Obsah

Seznam použitých zkratek a symbolů

AWS	–	Amazon Web Services
AMI	–	Amazon Machine Image
API	–	Application Programming Interface
CRUD	–	Create Remove Update Delete
DNS	–	Domain Name System
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
IČO	–	Identifikační Číslo Organizace
JAR	–	Java Archive
JPA	–	Java Persistence API
JPQL	–	Java Persistence Query Language
JSON	–	JavaScript Object Notation
NVM	–	Node Version Manager
ORM	–	Object Relational Mapping
REST	–	Representational State Transfer
SQL	–	Structured Query Language
SSD	–	Solid State Drive
SSH	–	Secure Shell
SSL	–	Secure Sockets Layer
SSR	–	Server Side Rendering
URI	–	Uniform Resource Identifier
UUID	–	Universally Unique Identifier
VPC	–	Virtual Private Cloud

Seznam výpisů zdrojového kódu

1 Úvod

Během svého studia jsem se při výběru bakalářské práce rozhodl využít příležitost a v rámci ní jsem si vybral absolvování individuální odborné praxe ve společnosti Tieto Czech s.r.o. Hlavním důvodem k tomuto rozhodnutí bylo získání nových praktických zkušeností z oblasti vývoje softwaru a prohloubení stávajících znalostí získaných na vysoké škole. Zároveň mě při této příležitosti lákalo poznat firemní prostředí a osvojit si některé často využívané praktiky a postupy při vývoji softwaru.

Za tímto účelem jsem do společnosti nastoupil jako stážista na pracovní pozici Software Developer, během níž jsem se podílel na vývoji nového webového portálu Dobromila, který měl primárně umožnit navazování spoluprací s neziskovými organizacemi. Při této příležitosti jsem měl možnost získat řadu nových zkušeností se samotným vývojem aplikace i s pozdějším nasazením do produkčního prostředí a při práci s cloudovými službami Amazon Web Services (AWS).

Samotná práce je rozdělena na několik hlavních částí. V první z nich se věnuji krátkému popisu společnosti a mému pracovnímu zařazení. Další obsahuje detailní popis projektu, na němž jsem se podílel, jeho prvotní přípravy a výběru technologií, které jsem při vývoji využil. Další části jsou věnovány podrobnému popisu zadaných úkolů, jejich řešení, včetně problémů se kterými jsem se v jejich průběhu setkal. Na závěr shrnuji využité znalosti z vysoké školy, celkový průběh praxe a zhodnocuji dosažené výsledky.

2 O společnosti Tieto Czech s.r.o.

2.1 Odborné zaměření firmy

Tieto je největší skandinávskou IT společností, která zákazníkům nabízí kompletní IT řešení v širokém spektru odvětví, mezi něž patří telekomunikace, bankovnictví a pojišťovnictví, výrobní průmysl, veřejná správa nebo například strojírenství a lesnictví.

Celosvětově společnost působí v téměř 20 zemích včetně České republiky a zaměstnává dohromady 15 000 lidí. U nás tato společnost ve svých pobočkách v Ostravě a Brně zaměstnává více než 2600 lidí, z nichž až 200 každý měsíc cestuje. [1] [2] [3]

2.2 Pracovní zařazení

Po splnění všech předpokladů jsem do firmy nastoupil jako stážista na pracovní pozici Software Developer. Mezi moje zaměření patřil především vývoj v jazyce Java s hlavním využitím moderního frameworku Spring Boot. Během stáže jsem měl ale příležitost se také podrobněji setkat s několika dalšími technologiemi a knihovnami jako například s ORM frameworkem Hibernate, relační databází PostgreSQL, dokumentovou databází MongoDB, vyhledávacím enginem Elasticsearch nebo s cloudovými službami Amazon Web Services (AWS). Mimo jiné jsem se setkal s agilním vývojem za použití metodiky Scrum nebo také s využitím unit testů.

Oddělení, ve kterém jsem pracoval, bylo složeno z profesionálních vývojářů i ze stážistů, kteří se podíleli především na implementaci řešení určených pro interní využití ve společnosti. Díky této kombinaci nebyl problém v případě potřeby cokoliv diskutovat se zkušenějšími vývojáři. Této příležitosti jsem několikrát využil, a to například při přípravě produkčního prostředí nebo při integraci celého projektu s cloudem. V rámci této možnosti jsem získal nové zkušenosti, díky čemuž jsem byl později schopen řešit podobné úlohy sám.

3 Webová aplikace Dobromila

V rámci mého působení na stáži, se mi naskytla možnost pracovat na novém začínajícím projektu pro externího zákazníka. Byla to jedinečná příležitost si vyzkoušet kompletní realizaci aplikace, která zahrnuje jak jednání se zákazníkem, tak i samotnou realizaci a předání produktu. Této příležitosti jsem využil a se svými kolegy jsme utvořili projektový tým, který se na realizaci podílel. Náš tým čítal čtyři lidi včetně mě, přičemž dva z nás se podíleli na backendové části aplikace a zbylí dva na frontendové části. Vzhledem ke svému zaměření, jsem se podílel zejména na backendové části a později i na nasazení celého projektu do cloudového prostředí.

3.1 Popis projektu

Cílem projektu bylo vytvořit webovou aplikaci, která by umožnila jednoduché navázání spolupráce mezi organizacemi a společnostmi z neziskového, veřejného a soukromého sektoru. Těmto subjektům by měla aplikace usnadnit nabízet nebo poptávat různé služby, vyhledávat a objevovat nabídky nebo poptávky jiných subjektů, navazovat s nimi spolupráce a v neposlední řadě prezentovat se na veřejnosti prostřednictvím svého profilu. K tomuto účelu by měl sloužit přehledný katalog nabídek a poptávek, katalog organizací a sekce pro registrované uživatele. Oba katalogy by zároveň měly být dostupné veřejnosti bez nutnosti registrace a měly by nabízet pohodlné fulltextové vyhledávání s možností filtrování výsledků podle různých kritérii.

Příkladem využití by mohla být situace, ve které organizace vlastnící nepotřebný, avšak použitelný nábytek, mohla využít portál k tomu, aby ho nabídla jiným organizacím nebo firmám. Během tvorby nabídky si v tomto případě může organizace zvolit druh vyrovnání, který je pro ni výhodný, lokalitu a příslušnou kategorii. Potencionální zájemci by měli možnost nabízející organizaci skrze portál kontaktovat a posléze s ní zahájit spolupráci. Organizace, která nabídku vytvořila, by následně měla možnost spolupráci přijmout nebo odmítnout podle vlastního uvážení. Úspěšně dokončenou spolupráci by mělo být možné oboustranně ohodnotit přímo v aplikaci. Součástí hodnocení by také měl být, kromě stupnicového vyjádření, i komentář poskytující zpětnou vazbu.

3.2 Příprava projektu

Práce na projektu jsme započali schůzkou se zákazníkem, který nám sdělil svou představu o aplikaci, její účel a funkční požadavky. Během našeho sezení jsme postupně se zákazníkem rozvíjeli detaily systému, upřesňovali požadavky, které by aplikace měla splňovat, a hledali jejich nejvhodnější řešení. Mezi prvními tématy byl například základní princip aplikace a její zaměření včetně možností využití. Zde jsme se bavili o tom, komu je aplikace určena, jak by mu měla pomoci, co by mu měla poskytovat, kde by nalezla své využití a jakým způsobem by měla aplikace plnit svoji funkci. Při této diskuzi jsme zároveň řešili funkční a nefunkční požadavky a detaily některých funkcí jako například vyhledávání, historizace dat, možnosti registrace apod.

Mimo to jsme se během naší diskuze dostali i k otázce designu a rozvržení aplikace. U tohoto tématu jsme se zabývali hlavně tím, jak na uživatele zapůsobit, jakým způsobem by měl vypadat profil organizace, co by na takovém profilu mělo být nebo jak by měla vypadat karta s detailem nabídky nebo poptávky. Zde byl ze strany zákazníka kladen důraz hlavně na moderní design, přehlednost a správnou prezentaci organizací. Na základě těchto představ, jsme později připravili grafický návrh celé aplikace, který byl zároveň hlavním tématem naší další schůzky.

V návaznosti na to jsme pokračovali tvorbou datového modelu. U tohoto tématu nás nejvíce zajímalo, jaké data by měly být uchovávány a jaké by měly mezi nimi být vztahy a souvislosti. Například v rámci nabízení či poptávání služeb/produktů, by měla aplikace umožňovat definovat kromě základního popisu i kategorie (Odborná služba, Vybavení apod.), druh vyrovnání (platba, výměna apod.) nebo také lokalitu, ve které je možné danou službu uplatnit. Pro lepší reprezentaci bylo zároveň požadováno, aby bylo možné nahrát i skupinu obrázků popisující danou službu či produkt. Vzhledem k těmto požadavkům, jsme probírali, jak by mělo být nahrávání obrázků omezeno (velikostně, počtem) a které z atributů jsou povinné.

Nejvíce času jsme ale strávili diskuzí ohledně funkcionalit, které by aplikace měla mít. Jedním z požadavků byla nutnost schválení registrovaných organizací a společností správcem. To by mělo zabránit tomu, aby se do katalogu dostaly podvodné organizace. Po registraci by profil organizace neměl být veřejný a zároveň by nově vytvořená organizace neměla mít možnost vytvářet nabídky a poptávky a ani by neměla mít možnost na některou z nich reagovat.

Celou naši schůzku jsme nakonec zakončili naplánováním dalších kroků a termínů, ve kterých bychom náš postup konzultovali.

3.3 Výběr technologii

Projekt jsme se rozhodli postavit na dvou hlavních platformách, a to na Javě a JavaScriptu, které nejenomže splňují všechny nároky na moderní vývoj enterprise aplikací, ale jsou také dlouhodobě využívané jako osvědčené řešení při vývoji rozsáhlých systémů. Kromě těchto platforem jsme se rozhodli využít několik dalších pomocných technologií, knihoven a frameworků, které nám během vývoje usnadnily spoustu práce. Mezi ty nejdůležitější bych si dovolil zmínit framework Spring spolu s navazujícím projektem Spring Boot, Hibernate, Elasticsearch, MongoDB a cloudové řešení AWS. Jejich stručný popis spolu s příklady využití v našem projektu jsou uvedeny níže.

Spring Boot – nástavba umožňující jednoduché vytvoření samostatné aplikace založené na Spring frameworku. Výhodou je v tomto ohledu jednoduchá konfigurace a sada předdefinovaných balíčků poskytujících základní funkce.

Elasticsearch – vyhledávací a analytický nástroj, určený k uchování, vyhledávání a analýze velkých objemů dat. Jeho hlavní předností jsou široké možnosti nastavení vyhledávání a filtrace, rychlost a spolehlivost. V naší aplikaci jsme se rozhodli využít Elasticsearch především pro vyhledávání mezi organizacemi, nabídkami a poptávkami. Hlavním důvodem

k volbě tohoto nástroje byla především zmíněná rychlost vyhledávání i pro velký počet aktivních uživatelů a možnost škálování do budoucna. Mimo to nás přesvědčila i jednoduchá integrace s použitým frameworkem Spring Boot.

MongoDB – dokumentová databáze umožňující flexibilní uchování dat. Výhodou této databáze je uchování záznamů v podobě dokumentů s volnou strukturou. V praxi to znamená, že struktura jednotlivých záznamů se může lišit od ostatních. Této vlastnosti je v naší aplikaci využíváno například při uchování změn profilů jednotlivých firem a organizací pro účely schválení.

AWS – cloudová platforma poskytující IT infrastrukturu spolu s širokou škálou služeb od výpočetní kapacity, přes databázové úložiště, síťovou infrastrukturu až po nejrůznější vývojářské nástroje. Mezi hlavní výhody patří vysoká dostupnost a spolehlivost služeb, platba pouze za využité prostředky (výpočetní čas, kapacita) nebo třeba možnost postavit kompletní řešení pod jedním cloudovým dodavatelem.

Hibernate – ORM framework implementující specifikaci Java Persistence API (JPA). Jeho přítomnost v aplikaci usnadňuje perzistentní mapování objektových dat na relační strukturu. V našem projektu je využit jako hlavní nástroj pro přístup k databázi a pro práci s daty.

QueryDSL – podpůrná knihovna poskytující doménově specifický jazyk pro tvorbu SQL a JPQL dotazů. Hlavní výhodou je pohodlný zápis dotazů pomocí sady tříd a metod, které ve výsledku zajišťují přehledný a dobře čitelný kód zejména u komplikovanějších dotazů. Kromě toho umožňuje i dynamickou skladbu dotazů, kterou jsem rovněž využil.

4 Seznam zadaných úkolů

Během své praxe jsem dostával úkoly spojené zejména s prací na projektu webové aplikace Dobromila. Vzhledem k tomu, že se jednalo o nový projekt, mé úkoly se ze začátku týkaly nejprve návrhu architektury, perzistentní vrstvy a business logiky. Poté jsem se přesunul k implementaci a jako poslední věc mě čekalo testování a výsledné nasazení do produkčního prostředí.

K vedení a správě všech úkolů, které byly potřeba na projektu udělat, jsem využíval nástroj GitLab Issues. Ten je součástí aplikace GitLab, který jsem rovněž využil pro správu verzí zdrojových kódů projektu. Pomocí tohoto nástroje jsem si plánoval dopředu úkoly, na kterých budu pracovat, uváděl u nich odhadovanou časovou náročnost a patřičné tagy. Naplánované úkoly jsem poté podle toho, v jakém stavu byly, přesouval mezi sloupci *Backlog*, *In progress* a *Closed*. Tyto sloupce v uvedeném pořadí označovaly úkoly, které bylo potřeba udělat, právě se na nich pracovalo nebo už byly dokončené. Všechny úkoly jsem dále ještě řadil do tzv. *milestone*, které udávaly aktuální *sprint*. Většinou byly tyto *milestone* postavené na měsíční bázi.

4.1 Časová náročnost

Na své praxi jsem její podstatnou část strávil vývojem samotné webové aplikace Dobromila. Tato část zahrnovala jak návrh, tak i samotnou implementaci řešení. Věci spojené s tímto vývojem mi zabraly odhadem 40 dní. Během této části probíhala zároveň příprava testovacího prostředí, které mělo sloužit jako demo pro zákazníka. Vzhledem k tomu, že jsem neměl příliš zkušeností s cloudem a s nástroji od AWS, zabrala mi tato část zhruba 14 dní.

5 Návrh a implementace projektu

5.1 Návrh struktury projektu

Práci na projektu jsme zahájili návrhem jeho struktury. Nejprve jsme se věnovali základnímu rozdělení systému a dále pak jeho podrobnějšímu dělení, strukturalizaci a architektuře. To znamenalo, že jsme si celý systém rozdělili na backendovou a frontendovou část a dále se věnovali návrhu právě těchto částí. Ve své práci se dále budu soustředit pouze na backendovou část, na jejímž vývoji jsem strávil nejvíce času a jejíž strukturu jsem také navrhoval.

Jako první jsem začal rozložením této části do dvou modulů. K tomu jsem využil nástroj Maven, který toto dělení umožňoval. Prvním z takových modulů byl modul *auth-service*, jehož účelem bylo poskytnutí možnosti autentifikace a autorizace přístupu k jednotlivým částem celého systému. Jeho princip spočíval v uchování autentifikačních a autorizačních údajů a poskytnutí svých služeb dalším modulům, jako byl například i druhý modul *offering-service*. Ten se už, na rozdíl od modulu *auth-service*, věnoval implementaci hlavních částí aplikace, které byly spojeny s tvorbou nabídek a poptávek, s navazováním spoluprací, s jejich hodnocením ale i se správou účtů.

Struktura tohoto modulu odpovídala standardní struktuře Maven projektu, kde nejdůležitější složkou byla */src/main/java*. Ta následně uchovávala kompletní zdrojový kód aplikace rozdělený mezi jednotlivými balíčky, které na té nejvyšší úrovni odpovídaly logickým celkům spojených s vrstvami mnou zvolené třívrstvé architektury a dalším součástem aplikace. Na jejich nižších vrstvách jsem dále tvořil další balíčky podle potřeby, a to zejména tak, aby pokud možno sjednocovaly nějaký logický celek. Příkladem takového dělení může být balíček *repository*, který v sobě uchovával třídy, rozhraní a výčty spojené s perzistentní vrstvou. Ty se dělily nejprve podle toho, s jakým typem databáze jsou dané elementy spojeny a dále i podle toho o jaký typ třídy jde, tedy jestli se jedná o entitu, výčet nebo jiný element.

Kromě zdrojového kódu, byly součástí aplikace i další nezbytné soubory, přičemž jejich hlavní umístění bylo ve složce */src/main/resources*. Ty mezi sebou zahrnovaly konfigurační soubory aplikace, mailové šablony nebo inicializační skripty, které v aplikaci rovněž hrály důležitou roli.

5.2 Návrh perzistentní vrstvy

Jedním z mých úkolů bylo navrhnout a implementovat perzistentní vrstvu. Začal jsem nejprve výběrem vhodné databáze, popř. databází, které by pokryly všechny požadavky zadané zákazníkem. Vzhledem k těmto požadavkům jsem se nakonec rozhodl využít celkem tři databáze. První z této trojice byla databáze relační, která je zároveň i tou nejdůležitější, jelikož by měla sloužit jako hlavní úložiště většiny dat. Mezi tyto data například patří všechny záznamy o uživatelských účtech, nabídkách, poptávkách, spolupracích nebo hodnoceních. Další z databází, kterou jsem se rozhodl využít, byla databáze dokumentová. Její hlavní přínos spočíval v uchování záznamů o změnách jednotlivých účtů a v uložení metadat o obrázcích, které by rovněž měly být v aplikaci

využity. A nakonec jako poslední z trojice jsem využil databázi pro vyhledávání, která měla sloužit zejména pro rychlé fulltextové vyhledávání s možností komplexní filtrace. Vzhledem k povaze této databáze, byly ale záznamy v ní pouhou kopií těch uložených v databázi relační. Jednalo se tak především o záznamy společensky prospěšných organizací, firem, nabídek a poptávek.

Po rozmyšlení účelů jednotlivých databází jsem se postupně pustil do jejich návrhu. Jako první jsem začal databází relační. Návrh celého modelu pro tuto databázi jsem si rozdělil na několik částí podle zaměření. V první části jsem řešil detaily uložení informací o jednotlivých uživatelských účtech. Vzhledem k požadavku umožnit registraci neziskovým organizacím, firmám i fyzickým osobám bylo zapotřebí vést informace o třech různých typech účtů. K tomuto účelu jsem si vytvořil celkem čtyři tabulky, každou pro jeden typ účtu a jednu další, která sloužila k uchování společných dat. Podřízené tabulky konkrétních typů účtů jsem následně navázal na společnou tabulku pomocí cizího klíče. Ten zároveň v podřízených tabulkách hrál roli primárního klíče. Díky tomu byla zajištěna návaznost a unikátnost napříč těmito tabulkami. Velkou výhodou tohoto řešení je možnost smazání pouze záznamů konkrétních profilů, s možností zachování nadřazených záznamů. To se hodí například při zrušení účtu, kdy chceme detaily profilu smazat, avšak nechceme porušit vazbu mezi spolupracemi a hodnoceními, které daný uživatel vytvořil.

Ve druhé části jsem se věnoval tvorbě tzv. číselníků. Ty by měly představovat tabulky sloužící pro uchování dat týkajících se typu služby, typu kompenzace, lokality nebo kategorie. Struktura těchto tabulek je obvykle stejná a zahrnuje pouze identifikátor a název. Výjimku tvoří pouze tabulka pro uložení lokalit, ve které se mimo jiné nachází i reference na související záznam regionu.

Jako poslední část jsem řešil návrh tabulek zajišťující uchování záznamů spojených s nabídkami a poptávkami. Před samotným návrhem si ale nejprve bylo nutné ujasnit význam nabídky a poptávky v kontextu naší domény, jejich vlastnosti a způsob použití. Například nabídky v námi používaném kontextu slouží k nabídnutí zboží či služeb zákazníkovi, a to darem nebo výměnou za finanční hotovost, popřípadě za jinou službu. Poptávky naopak slouží k vyjádření zájmu o nějaké zboží či službu definovaných parametrů. Z pohledu vlastností, jsou pak nabídky i poptávky velmi podobné a mohly by být reprezentovány jedinou entitou. Nicméně jsem se rozhodl je i v aplikační doméně ponechat rozdělené, protože při pozdějším vývoji s přibývajícími požadavky, by sloučení entit mohlo zkomplikovat celý proces implementace. Kromě myšlenky na budoucí použití, rozdělení do dvou tabulek by mohlo přinést výhodu i v podobě zvýšení výkonu při dotazování na databázi, neboť vyhledávání se bude obvykle uskutečňovat pouze nad částí dat.

S nabídkami a poptávkami jsou dále spojeni kandidáti a jejich hodnocení. Kandidátem nabídky či poptávky se v tomto případě rozumí zájemce o spolupráci. Záznam pro takovou spolupráci obsahuje zejména informace o účtu, nabídce či poptávce a krátkou textovou poznámku. V té může uživatel popsat své dodatečné požadavky na nabídku nebo poptávku, popřípadě uvést jiné sdělení. Nedílnou součástí jsou také hodnocení, které se vážou na uzavřené spolupráce. Jejich obsahem je kromě stupnicového hodnocení i krátký komentář a informace o autorovi.

Po dokončení návrhu posledních tabulek relační databáze jsem se pustil do návrhu kolekcí databáze dokumentové. I přestože struktura dat dokumentové databáze není pevná, bylo alespoň nutné si určit, jaká data se do ní budou ukládat a v jakých kolekcích budou uchovány. Ze začátku nám postačilo do této databáze ukládat pouze záznamy se změnami účtů, které byly potřeba uchovávat hlavně z důvodu nutnosti verifikace a následného schválení ze strany správce webu. K tomuto účelu nám postačila jedna kolekce, která umožňovala uložení všech typů účtů. Mezi jejími atributy bychom následně mohli najít kombinaci všech atributů ze všech typů účtů.

5.3 Implementace perzistentní vrstvy

Perzistentní vrstvu jsem se rozhodl implementovat v tom pořadí, v jakém jsem ji navrhl. Začal jsem proto relační databází a pro práci s jejími daty jsem využil specifikaci Java Persistence API (JPA). Jako implementaci jsem si vybral dnes již velmi populární ORM framework Hibernate, který v rámci specifikace umožňuje vytvářet objektově relační mapování. Tuto vlastnost jsem využil a pro tyto účely jsem začal postupně tvořit entitní třídy, které odpovídaly jednotlivým tabulkám v databázi.

První takovou třídou byla třída `AccountDO`, jejíž instance představovala jeden řádek z tabulky sloužící pro uchování informací o účtech. Po jejím vytvoření bylo nutné nad tuto třídu umístit anotaci `@Entity` a deklarovat atribut identifikátoru, který byl označen anotací `@Id`. Jako atribut identifikátoru jsem v tomto případě zvolil umělý primární klíč s datovým typem `UUID`.

Použití datového typu `UUID` jako primárního klíče spolu nese několik výhod. Mezi ně zejména patří to, že takto vygenerované klíče jsou v podstatě unikátní napříč vícero systémy. Je tedy menší pravděpodobnost, že bychom narazili na problém s duplikátními klíči při dodatečném importu nebo synchronizaci dat. Unikátnost vygenerovaných hodnot je nicméně ovlivněna tím, jakou verzi UUID použijeme. Těchto verzí je celkem čtyři, přičemž `UUIDGenerator` v námi použitém frameworku Hibernate podporuje dvě a to verzi 1 (RFC 4122 Version 1) a verzi 4 (RFC 4122 Version 4). Jako výchozí strategie je použita verze 4, což znamená, že generování identifikátoru je založeno na náhodných číslech. Náhodné generování spolu přináší i několik dalších výhod, mezi něž by se dala zařadit například menší šance na uhodnutí oproti sekvenčně generovaným identifikátorům. Oproti číselným identifikátorům, ale nese použití UUID zvýšené nároky na prostor potřebný k uložení, jelikož má vygenerovaný klíč délku 128 bitů. V našem případě to nicméně nevadí, protože počet záznamů, které budou v aplikaci uloženy, nebude pravděpodobně tak velký, aby délka primárního klíče pro nás znamenala vážný problém. Navíc z hlediska podpory nabízí námi zvolená databáze PostgreSQL přímo vhodný datový typ a podporu pro uložení a práci s klíči UUID. [4]

Proto, aby automatické generování fungovalo, bylo ale nejprve nutné zaregistrovat odpovídající generátor. K tomuto účelu sloužila anotace `@GenericGenerator` v níž jsem definoval název a strategii pomocí které se budou nové hodnoty generovat. V mém případě jsem jak pro název, tak i pro strategii použil hodnotu *uuid2*. Až po tomto kroku jsem mohl nově zaregistrovaný generátor použít v anotaci `@GeneratedValue`, ve které jsem v atributu `generator` uvedl jeho

název. Tuto anotaci jsem rovněž musel umístit nad identifikátor, tak aby bylo zřejmé, že při uložení nového záznamu má dojít k vygenerování primárního klíče.

Dále jsem pokračoval definicí zbytků atributů, z nichž za zmínku stojí určitě atribut pro určení stavu, ve kterém se účet aktuálně nachází. Těchto stavů může v aplikace nastat několik a patří mezi ně například stav označující nově vytvořený účet nebo stav označující to, že byl účet schválen či zamítnut. Tyto stavy jsou zejména důležité pro řízení životního cyklu celého účtu, což pokrývá také požadavek na nutnost každý účet včetně jeho změn nejprve schválit. Pro tuto potřebu jsem si v aplikaci vytvořil výčet, jehož hodnoty označují všechny možné stavy, které u účtu mohou nastat. Díky tomuto výčtu jsem se mohl nadále na jednotlivé konstanty odkazovat přímo z kódu, aniž bych je musel ručně uvádět pomocí řetězců. Pro jejich správné uložení bylo ale nejprve nutné do entity uvést, jakým způsobem má být takováto hodnota uložena. K tomu mi posloužila anotace `@Enumerated` a její jediný atribut `value`, pomocí něhož jsem určil, zda daný atribut má být uložen buď jako číselná hodnota nebo jako řetězec. V případě číselné hodnoty se jednalo o index výčtové konstanty, který ovšem závisel na pořadí, v jakém byly konstanty definovány. Toto řešení sice má za následek menší paměťové nároky, avšak změna pořadí konstant může mít v tomto případě fatální následky, jelikož indexům budou posléze odpovídat jiné hodnoty. Já si proto vybral řetězcovou reprezentaci a do anotace jsem vložil odkaz na konstantu `EnumType.STRING`. Obdobný případ s výčty mě ještě čekal i atributu specifikující uživatelskou roli. Zde jsem postupoval stejně a daný atribut jsem se opět rozhodl ukládat jako řetězec. [5]

V rámci anotování modelu mi dále ještě zbývalo umístit anotaci `@Column`, což byla také jednou z nejčastěji použitých anotací v entitách. Jejím hlavním cílem je definovat, jakým způsobem mají být jednotlivé atributy mapovány. K tomuto účelu poskytuje tato anotace několik atributů, z nichž jsem nejčastěji využil atribut `name` pro určení názvu sloupce, `nullable` pro určení povinnosti sloupce, `unique` pro definici omezení spojeného s unikátností hodnot a atribut `length` pro určení maximální velikosti hodnot.

Další entitní třídou, na kterou jsem navázal, byla třída `OrganizationDO`. Ta sloužila pro uchování dat o profilech společensky prospěšných organizací. Stejně jako u třídy `AccountDO`, i zde jsem umístil nad tuto třídu anotace `@Entity` a `@Table`. Změna ale nastala při definici atributu primárního klíče. Ten totiž musel být zároveň cizím klíčem, což vyžadovalo speciální způsob mapování. K tomuto účelu jsem si vytvořil atribut primárního klíče mající stejný datový typ jako primární klíč ve třídě `AccountDO` a umístil nad něj anotaci `@Id`. Místo toho ovšem abych pomocí anotace `@GeneratedValue` zajistil automatické generování, tak jsem definoval další atribut, který se odkazoval na instanci typu `AccountDO`. Nad tento atribut dále již jen zbývalo umístit anotaci `@MapsId`, která indikovala to, že primární klíč entity odpovídá primárnímu klíči instance daného atributu a anotaci `@OneToOne` označující atribut za cizí klíč. Zároveň jsem zde přidal ještě poslední anotaci `@JoinColumn`, která plní podobnou funkci jako anotace `@Column`, ale pro cizí klíče, a definoval v ní název sloupce.

Ze zbytku atributů třídy `OrganizationDO`, bych se rád ještě zaměřil na kolekci kategorií,

která je zajímavá tím, že se u ní neprovádí přímé mapování na jeden z databázových atributů entity, v němž je definována. Místo toho ale představuje jednu ze stran obousměrné vazby M:N, přičemž mapování této vazby je provedeno podle vzoru Association Table Mapping. K tomuto mapování jsem opět využil anotace, a to konkrétně `@ManyToMany` a `@JoinTable`.

Anotace `@ManyToMany` mi zde posloužila pouze k označení toho, že se atribut váže ke vztahu M:N, přičemž tento vztah jsem následně definoval zmíněnou anotací `@JoinTable`. Pomocí ní je možné určit název spojovací tabulky a definici klíčů, které se odkazují na klíče navázaných tabulek. Ukázka definice takového vztahu je poté vidět ve výpisu 1.

```
@ManyToMany
@JoinTable(
   name = "organization_categories",
   joinColumns = @JoinColumn(name = "organization_id", nullable = false),
   inverseJoinColumns = @JoinColumn(name = "category_id", nullable = false)
)
private Set<CategoryDO> categories;
```

Výpis 1: Použití anotace @ManyToMany

Posledními entitami, které jsem v aplikaci implementoval, byly ty, které se mapovaly do dokumentové databáze a na index Elasticsearch. Zde byla situace o dost jednodušší, jelikož entit bylo jenom pár a pro jejich mapování nebylo potřeba definovat vztahy ani jiná databázová omezení. Jediný problém nastal při tvorbě entit pro index Elasticsearch, kde bylo potřeba zvolit odlišných datových typů a obecně jiné reprezentace některých dat, a to hlavně z důvodu přizpůsobení k vyhledávání. Takové změny se například týkaly modelů `SupplySO` a `DemandSO` určených k ukládání dat pro vyhledávání nabídek a poptávek. V těch musely být oproti jejich vzorům `SupplyDO` a `DemandDO`, upraveny odkazy na navazující entity tak, že místo nich modely obsahovaly pouze jejich identifikátory. Ty poté šly zároveň jednoduše použít při vyhledávání, jelikož ve většině případů se tyto identifikátory odkazovaly na hodnoty číselníků.

Po dokončení implementace poslední entity mi zbýval vyřešit už jen poslední problém a tím byl audit. Auditing entit měl zajistit to, aby se spolu s daty jednotlivých entit ukládala i informace o tom, kdy došlo k vytvoření nebo úpravě záznamu a kdo takovou změnu naposledy provedl. K tomuto účelu jsem využil možnosti posluchače událostí, který byl schopen reagovat na jednotlivé události v rámci životního cyklu entity. Takovou událostí může být například uložení, aktualizace nebo smazání záznamu. Pro tyto účely následně slouží odpovídající anotace jako `@PrePersist`, `@PreUpdate` nebo `@PreRemove`, z nichž jsem nakonec dvě ve svém posluchači využil. Aby ale posluchač událostí mohl reagovat na dané událost, bylo ho potřeba nejprve zaregistrovat. K tomu sloužila anotace `@EntityListener`, kterou bylo potřeba umístit nad danou entitu. Součástí této anotace byl jediný atribut `value` definující pole posluchačů událostí, které mají být pro danou entitu použity. Mě nicméně postačil pouze jeden posluchač, v němž jsem defi-

noval metody `setCreationDetail` a `setModificationDetail` nad něž jsem umístil již zmíněné anotace `@PrePersist` a `@PreUpdate`. Těmto metodám byl následně vždy předán jeden argument představující entitu, v níž ke změně došlo. Jelikož jsem ale použil jeden obecný posluchač událostí, musel jsem v metodách pracovat buď s parametrem typu `Object` nebo v lepším případě s parametrem typu nějakého rozhraní. Já se přiklonil k druhé variantě, jelikož jsem se tímto chtěl vyhnout kontrole typu a následnému přetypování, a vytvořil jsem si rozhraní `Auditable`. V něm jsem následně definoval hlavičky dvou metod – `setAudit` a `getAudit`, které mi umožňovaly jednoduchou manipulaci s auditními daty ještě před uložením změn do databáze. Takovéto rozhraní mi už jen stačilo implementovat ve všech entitách, které vyžadovaly ukládání auditních informací.

Samotné ukládání a mapování auditních dat jsem dále vyřešil pomocí návrhového vzoru Embedded Value. Ten umožňuje provést mapování hodnot jednotlivých atributů objektu na odpovídající atributy záznamu v databázové tabulce, která náleží entitě obsahující daný objekt. Díky tomu je možné odkazovat se a pracovat se skupinou souvisejících atributů snadněji, než kdybychom s nimi měli jednotlivě pracovat. Dále se využitím tohoto vzoru snižuje redundance atributů, jelikož entit, které vyžadují ukládání auditních informací je v aplikaci několik a tím, že tyto atributy definuji pouze jednou, předcházím jejich opakování.

Třída zapouzdřující auditní informace vypadá obdobně jako entita, až na několik rozdílů. Tím prvním je to, že nad třídou se nachází pouze jediná anotace a to `@Embeddable`, která značí, že obsažené atributy se mají mapovat na položky tabulky entity, ve které se nachází instance této třídy. Druhým rozdílem pak je absence definice primárního klíče. Ten zde není nutný, a to především z toho důvodu, že instance této třídy netvoří samostatné záznamy v databázi. Co zde ovšem zůstává společné jako v entitě, je možnost k atributům definovat jakým způsobem se mají mapovat, a to pomocí anotace `@Column`.

Výsledné použití v jednotlivých entitách bylo nakonec jednoduché a jediné co mi stačilo udělat, bylo přidat atribut s typem mnou vytvořené třídy a umístit nad něj anotaci `@Embedded`. Ta následně značí, že atributy, které se mají mapovat, jsou umístěny v daném objektu.

5.4 Implementace vrstvy přístupu k datům

Po dokončení tvorby entit jsem pokračoval implementací vrstvy přístupu k datům. Pro tyto účely jsem využil zejména framework Spring Data. Ten má za cíl snížit množství kódu potřebného pro implementaci perzistentní vrstvy bez ohledu na typ databáze. Díky něj jsem nemusel ručně implementovat jednotlivé CRUD operace a některé jednodušší dotazy.

Pro základní využití stačí pouze vytvořit pro každou entitu patřičné rozhraní, které rozšiřuje jedno z předdefinovaných rozhraní tohoto frameworku. Takovým rozhraním může být například `Repository`, `CrudRepository` či `PagingAndSortingRepository`. Pro každé z těchto rozhraní následně stačí uvést typové parametry označující typ entity a typ primárního klíče. Takto vytvořené rozhraní poté disponuje množstvím metod závisejících na rozšířeném rozhraní. Například pokud rozšíříme rozhraní `PagingAndSortingRepository`, získáme přístup jednak k základním

metodám, které toto rozhraní dědí od `CrudRepository` ale také k jeho vlastním metodám. Z nich za zmínku určitě stojí přetížené metody `findAll`, umožňující načítat data z databáze po stránkách nebo vracet je seřazené podle zvolených kritérií. Díky tomu je možné pracovat vždy pouze s daty, které jsou při zpracování nebo zobrazení nezbytně nutné.[16]

Kromě předdefinovaných metod lze do rozhraní přidat i svoje vlastní, přičemž patřičnou implementaci můžeme přenechat frameworku Spring Data. Jedinou podmínkou, kterou tyto metody musí splňovat je to, že buď z jejich názvu lze přímo odvodit dotaz, nebo je nad jejich deklarací umístěna anotace `@Query`, která odpovídající dotaz obsahuje.

Odvození dotazu v případě první varianty je poměrně jednoduché a řídí se několika pravidly, jejichž podrobnější popis lze najít v dokumentaci. Já bych zde nicméně popsal alespoň jeden ze základních dotazů, který lze pomocí tohoto zápisu vytvořit a který jsem i v projektu následně využil. Tím je dotaz pro zjištění toho, zda existuje nějaký účet s danou emailovou adresou. Deklarace metody pro takový dotaz vypadala následovně:

```
boolean existsByEmail(String email);
```

Jak je vidět na příkladu, framework Spring Data je schopen dle jednotlivých částí názvu metody dotaz sestavit, předat mu parametry a jeho výsledek převést na typ určený návratovým typem metody.

Nicméně pro složitější dotazy je tento způsob definování dotazů poněkud nepraktický. Proto jsem práci s tímto frameworkem kombinoval ještě s jednou knihovnou a to QueryDSL. Ta mi umožnila požadované dotazy sestavit pomocí volání patřičných metod. K tomu abych mohl tuto knihovnu využít, jsem ale musel nejprve zajistit generování tzv. Q-entit, které sloužily jako dotazovací typy pro jednotlivé entitní třídy. Z tohoto důvodu jsem upravil konfigurační soubor *pom.xml*, tak aby při sestavení aplikace se tyto třídy vytvořily. Pro tento účel je zároveň dodáván spolu s knihovnou QueryDSL plugin, který toto generování zajišťuje.

Takto vygenerované třídy v sobě obsahují atributy odpovídající těm, které jsou obsaženy v entitní třídě, přičemž typy těchto atributů jsou zaměněny tak, aby obsahovaly metody představující operace nad nimi. Voláním těchto metod je poté možné například sestavit podmínku dotazu nebo třeba modifikovat jeho výsledek. K tomu ale abych mohl dotaz vůbec sestavit, bylo zapotřebí vytvořit si instanci třídy `JPAQueryFactory` a předat ji odkaz na objekt typu `EntityManager`. Pomocí takto vytvořeného objektu jsem následně mohl volat metody `select`, `from`, `where`, `orderBy` nebo třeba `groupBy`, tak abych výsledný dotaz sestavil. Praktická ukázka takového volání je vidět ve výpisu 2.

```
query
   .select(qSupply)
   .from(qSupply)
   .where(qSupply.creator.id.eq(accountId).and(...))
   .limit(page.getPageSize())
```

```
    .offset(page.getOffset())
    .fetchResults();
```

Výpis 2: Tvorba dotazu pomocí QueryDSL

Jak je vidět na ukázce ve výpisu 2, tak ani stránkované výsledky nejsou problém pomocí této knihovny získat. Stačí pouze použít metodu `limit` k omezení počtu vrácených výsledků a metodu `offset` k nastavení posunutí pozice v rámci řádků, od kterého budou výsledné záznamy vráceny. Tyto záznamy spolu s dalšími informacemi jako je celkový počet záznamů, jsou následně vkládány do konstruktoru třídy `PageImpl`, která implementuje rozhraní `Page`. Díky objektům této třídy lze klientům přes rozhraní REST vracet kromě dat i příslušné informace v podobě čísla stránky, celkového počtu záznamů nebo třeba celkového počtu stránek.

Nicméně při používání paginace jsem narazil i na jeden zajímavý problém, který se týkal okamžité inicializace entit spojených s jednou z variant klauzule JOIN. Tu lze zajistit buď připojením klíčového slova FETCH za klíčové slovo JOIN uvnitř definice JPQL dotazu nebo použitím jedné z příslušných metod knihovny QueryDSL. Každopádně použitím tohoto typu inicializace spolu s paginací, bude mít za následek to, že výsledné omezení počtu záznamů bude probíhat v paměti, o čemž jsem byl také informován upozorněním v konzoli aplikace:

```
HHH000104: firstResult/maxResults specified with collection fetch; applying in
memory!
```

Důvod proč nelze takovýto dotaz efektivně vykonat je ten, že Hibernate nedokáže plně definovat paginaci na úrovni SQL. Kvůli tomu musí Hibernate načíst všechny záznamy, které dotaz vrátí, a nad nimi poté provést paginaci ručně. To může u velkého množství záznamů způsobit značné výkonnostní problémy. Řešeních tohoto problému existuje několik, přičemž já si vybral to nejjednodušší. To spočívalo v tom, nevyužívat v případě paginace okamžitou inicializaci pomocí klauzule JOIN FETCH a místo toho pomocí anotace `@BatchSize` definovat množství záznamů, které se budou při inicializace vztahu načítat. Díky tomu se sice už neprovede pouze jeden dotaz, ale bude jich stále méně, než kdybychom všechny entity a vztahy inicializovali způsobem, který se označuje jako *„lazy“*, tedy až v případě vyžádání za běhu. Alternativou k tomuto řešení může být vykonání pomocí dvou dotazů nebo použitím tzv. *Window Functions*. [6] [7]

Kromě frameworku Spring Data, knihovny QueryDSL a obecně jazyka JPQL, jsem v aplikaci využil i nativní dotazy v SQL. Ty byly nutné použít celkem ve dvou případech, kdy nešlo JPQL použít. Mezi tyto případy patřilo například načtení záznamů nabídek a poptávek od populárních zadavatelů nebo načtení všech hodnocení udělených spolupracím u nabídek či poptávek od zvoleného účtu. V obou těchto případech bylo nutné, aby dotaz pracoval současně se záznamy dvou tabulek takovým způsobem, jako by se jednalo o data z jedné tabulky. To ovšem znamenalo, že data musela být sjednocena pomocí množinové operace UNION, která není jazykem JPQL podporována.

V takovém případě jsem využil možnost pracovat s nativními dotazy, k jejichž definici jsem použil tzv. pojmenované nativní dotazy. Ty se zapisují pomocí anotací `@NamedNativeQuery`, které se následně umístí nad některou z entitních tříd. K tomu abychom jich ale mohli definovat více, je nutné buď tyto anotace zaobalit do jedné anotace `@NamedNativeQueries` nebo pokud v projektu používáme Javu 8 a novější spolu se specifikací JPA 2.2 a novější, tak lze tyto anotace umisťovat za sebou přímo nad třídu. To je umožněno díky zavedení tzv. *repeatable annotations*, které přibyly právě ve verzi Java 8. V anotaci `@NamedNativeQuery` dále lze specifikovat několik atributů, přičemž nejpodstatnější pro mě byly atributy `name`, `query` a `resultSetMapping`. Pomocí nich lze dotaz pojmenovat, definovat a určit název příslušeného mapování, které bude použito při zpracování jeho výsledku. [8]

Takovéto mapování se opět definuje nad entitní třídu pomocí anotace `@SqlResultSetMapping`, přičemž zde platí stejné možnosti zápisu jako u anotace `@NamedNativeQuery`, až na to, že obalující anotací je v tomto případě `@SqlResultSetMappings`. Při samotné definici mapování stačí uvést jméno, třídu, na kterou se má mapování provést a samotné mapování. To může být realizováno například pomocí konstruktoru, kde stačí, aby výběr sloupců odpovídal patřičným atributům v jednom z konstruktorů zvolené třídy, a to včetně pořadí. Takto vytvořené mapování se poté automaticky použije při zpracování dotazu pomocí metody `createNamedQuery` zavolané na instanci třídy `EntityManager`. Tato metoda v případě přetížené varianty, kde je uvedena i třída, na kterou se má výsledek mapovat, vrací typově inicializovanou instanci generické třídy `TypedQuery`. Díky ní lze následně získat výsledek dotazu konkrétního typu. Příklad volání dotazu je uveden ve výpisu 3.

```
final Query queryCount = entityManager
   .createNamedQuery(AccountDO.SELECT_ACCOUNT_RATINGS_COUNT_BY_ACCOUNT_ID)
   .setParameter(ACCOUNT_ID, accountId);
[...]
final TypedQuery<RatingProjection> query = entityManager
   .createNamedQuery(AccountDO.SELECT_ACCOUNT_RATINGS_BY_ACCOUNT_ID,
      RatingProjection.class)
   .setParameter(ACCOUNT_ID, accountId)
   .setParameter(LIMIT_VAL, page.getPageSize())
   .setParameter(OFFSET_VAL, page.getOffset());
[...]
```

Výpis 3: Volání nativního dotazu

Jak je vidět v případě předchozí ukázky, proto abych mohl výsledky paginovat, tak je nutné volat ještě jeden dotaz navíc, který zjistí celkový počet všech záznamů odpovídající zadaným kritériím. Teprve až poté, co je ověřeno, že existuje alespoň jeden takový záznam, je spuštěn

další dotaz pro jeho získání. Současně jsou při tomto volání definovány paginační parametry jako velikost stránky a offset, díky kterým lze najít požadovanou stránku mezi výsledky.

V případě využití dokumentové databáze MongoDB je práce se Spring Data v podstatě totožná, až na několik menších změn. Mezi ty například patří použití jiného předdefinovaného rozhraní při tvorbě vlastních repozitářů, kde je v tomto případě využito rozhraní `MongoRepository` s typovými parametry pro typ dokumentu a typ jeho primárního klíče. Stejně jako u relační databáze zde ale zůstává stejná jak tvorba dotazů definovaných názvem metod v daném rozhraní repozitáře, tak i možnost použít anotaci `@Query` pro definici vlastního dotazu. Ten je ale nutno tvořit s ohledem na použitou databázi, přičemž v případě MongoDB je zápis dotazů značně odlišný od relační databáze a jazyka SQL.

I u databáze Elasticsearch je situace stejná. Taktéž jsem využil Spring Data, nicméně tentokrát s předdefinovaným rozhraním `ElasticsearchCrudRepository`. Rozdíl ovšem nastal u složitějších dotazů, kde jsem si už nevystačil s tím, co nabízí Spring Data, a tak jsem musel využít specifické rozhraní pro tvorbu dotazů dodané přímo od Elasticsearch. To mi umožnilo vytvořit komplexní dotazy pro hledání mezi nabídkami a poptávkami nebo organizacemi či firmami, a to podle nejrůznějších kritérií, jejichž výběr s následným kombinováním je možný skrze uživatelské rozhraní. Mezi takovými kritérii může být výběr několika kategorií nebo fulltextové hledání v názvu nebo popisku organizace. V takovém případě je dotaz sestaven na základě tzv. *vyhledávacího objektu*, který zvolené parametry obsahuje.

```
public Page<CompanySOImageWrapper> findAllByQuery(@NonNull CompanyQuery
    companyQuery, @NonNull Pageable page) {
   final String searchString = companyQuery.getSearchString();
   BoolQueryBuilder boolQuery = boolQuery();

   if (searchString != null) {
      boolQuery = boolQuery
         .must(
               disMaxQuery()
                  .add(matchPhrasePrefixQuery(CompanySO.FIELD_NAME, searchString
                     ))
                  .add(matchPhrasePrefixQuery(CompanySO.FIELD_DESCRIPTION,
                     searchString))
         );
   }
[...]
}
```

Výpis 4: Tvorba dotazu pro Elasticsearch

Na ukázce ve výpisu 4 je poté vidět jakým způsobem se takový dotaz skládá. V podstatě základem ve většině případů je využití třídy `BoolQueryBuilder`, pomocí něhož lze dotaz poskládat, a to třeba i z několika samostatných částí. K tomu slouží několik jeho metod, mezi nimiž je i metoda `must` nebo `should`, které určují povinnost. Do nich lze, obdobně jako v ukázce, vložit další část dotazu jako například statickou metodu `disMaxQuery`, která má za cíl výsledky poddotazů spojit pomocí množinové operace sjednocení. V mém případě by to znamenalo sloučení výsledků, jejichž název nebo popis začíná řetězcem předaným v objektu třídy `CompanyQuery` při volání metody.

Největší problém jsem ale zaznamenal při práci s dotazem, který pracuje se dvěma odlišnými indexy. Taková situace nastala při vyhledávání mezi nabídkami a poptávkami, kde největší obtíže způsobovalo mapování výsledků na odpovídající objekt. V takovém případě bylo nutné při volání dotazu uvést i objekt implementující rozhraní `ResultsExtractor`, který se o výsledné mapování staral. K vytvoření tohoto objektu jsem využil lambda výraz, v němž jsem volal vlastní metodu určenou pro mapování. V ní jsem následně musel výsledky postupně deserializovat z JSON formátu a převést je do objektu třídy `Page`.

Mezi dotazy by se ale dal najít i takový, který se od ostatních do jisté míry liší. Takovým dotazem je například ten, který slouží pro vyhledávání podobných nabídek a poptávek na základě existujícího vzoru. Pro jeho vytvoření stačí zavolat statickou metodu `moreLikeThisQuery` třídy `QueryBuilders` a do ní předat patřičné parametry jako je pole pro definování atributů, podle kterých se mají záznamy porovnávat a pole pro definování vzorových záznamů. Volání této metody poté vrací instanci třídy `MoreLikeThisQueryBuilder`, nad níž jdou volat další dodatečné metody sloužící pro parametrizaci chování dotazu. K nim ostatně také patřila metoda `minTermFreq`, pomocí níž lze určit minimální počet opakování termínů, které musí hledané záznamy splňovat.

```
   final MoreLikeThisQueryBuilder moreLikeThisQuery =
      moreLikeThisQuery(
         new String[]{
            SupplyDemandSO.FIELD_NAME, SupplyDemandSO.FIELD_DESCRIPTION
            },
         new MoreLikeThisQueryBuilder.Item[]{
            new MoreLikeThisQueryBuilder.Item(null, null, itemId.toString())
         })
         .minTermFreq(1)
         .minDocFreq(1)
         .maxQueryTerms(12);
}
```

Výpis 5: Tvorba dotazu pro vyhledávání podobných nabídek a poptávek

Na ukázce ve výpisu 5 je poté vidět, že vyhledávání podobných nabídek a poptávek probíhá podle shod v atributech názvu a popisu, přičemž vzorový záznam je určen identifikátorem. Své využití tento typ dotazu nalezne zejména při zobrazení podobných nabídek a poptávek u aktuálně zobrazené položky nebo při jejich personalizovaném doporučení.

5.5 Implementace aplikační vrstvy

Jedním z mých dalších úkolů na projektu bylo naimplementovat aplikační vrstvu. Ta v aplikaci hraje jednu z nejdůležitějších rolí, neboť se zde nachází veškerá business logika, která v aplikaci definuje nejrůznější pravidla, omezení a obecně i chování celého systému. Patří zde například pravidla a postupy spojené s registrací účtů a jejich schválením nebo se správou nabídek a poptávek. Kromě toho je zde zajištěn korektní přístup a práce s databází, včetně tvorby transakcí a komunikace s dalšími systémy v podobě mailového serveru nebo vybraných cloudových služeb. Všechny tyto pravidla se následně v projektu rozkládají napříč několika třídami, tzv. službami, zajišťující jednak práci s doménovými objekty, ale také s dalšími prostředky a komponentami. Přístup k těmto službám je poté možný prostřednictvím rozhraní, které každá ze služeb implementuje a zároveň poskytuje dalším službám nebo vyšším vrstvám.

Jednou z takovýchto služeb, kterou jsem implementoval, byla služba pro odesílání mailových notifikací. Ta měla za úkol odesílat maily na požadované adresy v závislosti na tom, k jaké události došlo. Takovou událostí může být schválení nově vytvořeného účtu či jeho změn nebo informace o tom, že jeho změny teprve čekají na schválení. Předtím než jsem se ale pustil do implementace samotných metod služby, bylo potřeba vyřešit to, jakým způsobem budou maily skládány. Na výběr se nabízelo několik řešení, přičemž jsem se nakonec rozhodl pro šablony, k čemuž mi později posloužila i knihovna Thymeleaf. Ta je ideální z toho důvodu, že poskytuje efektivní a jednoduchou práci s předdefinovanými HTML šablonami, které mohou být uloženy mimo zdrojový kód. Zároveň formát zpráv HTML umožňuje využívat i některé pokročilejší konstrukce, jako jsou tabulky a obrázky, díky čemuž mohou být zprávy obohaceny i o grafické prvky.

V návaznosti na to jsem si vytvořil ve složce *resources* další adresář pojmenovaný jako *templates*, do kterého jsem postupně vytvářel potřebné šablony. Ty vzhledem k používání speciálních atributů, musely mimo jiné také obsahovat odkaz na názvový prostor knihovny Thymeleaf, který tyto atributy definoval. Mezi takový atribut spadal i `th:text`, jehož hodnota udává klíč, podle něhož lze do těla dané značky dosadit odpovídající hodnotu. Díky němu bylo možné si nechat do šablony dynamicky dosadit e-mailovou adresu příjemce nebo další data. [9]

Po dokončení poslední šablony jsem jejich názvy spolu dalšími údaji umístil do konfiguračního souboru aplikace, tak aby bylo jednoduché tyto šablony v případě potřeby zaměnit za jiné. K jejich načtení jsem využil možnosti frameworku Spring Boot, který k danému účelu nabízí několik způsobů, jak s takovými daty pracovat. Já si vybral způsob, který umožňoval konfiguraci pohodlně spojit s proměnnými, a to za pomoci několika anotací `@Value`. Do nich mi postačilo umístit vždy kompletní cestu, která odpovídala stromu vlastností definovaném v konfiguraci.

Například pro získání názvu šablony e-mailu pro zaslání informace o schválení nového účtu, odpovídala tato cesta řetězci: *mail.templates.cs_CZ.account-approval.name*. Takto vytvořené a spojené proměnné jsem si seskupil do samostatné třídy `MailTemplateConfigHolder`, nad kterou jsem umístil anotaci `@Component`. Díky ní jsem si mohl nechat instanci dané třídy vytvořit v rámci inicializace aplikačního kontextu a posléze dosadit do třídy služby `MailServiceImpl` pomocí mechanizmu Dependency Injection. Tímto způsobem jsem si ostatně nechal do dané třídy dosadit i instanci třídy `TemplateEngine` a `JavaMailSender`, které mi umožňovaly zpracovávat připravené šablony a odesílat je pomocí e-mailu.

První metodou, ve které jsem tyto instance využil, byla metoda `sendAccountApprovalInfoEmail`, jejíž úkolem bylo odesílat notifikační e-maily o schválení či zamítnutí účtu. Pro tento účel v metodě sloužily dva parametry, přičemž první z nich udával e-mailovou adresu a druhý stav účtu. Následnou implementaci jsem poté začal přípravou kontextu, který byl nutný pro zpracování připravené šablony. K tomu mi posloužila instance třídy `Context`, kterou jsem si na začátku metody vytvořil a pomocí níž jsem definoval patřičné parametry pro dosazení. Tuto instanci jsem následně dosadil do metody `process` třídy `TemplateEngine`, která přijímala dva argumenty. Tím dalším už jen byl název šablony, přičemž jsem zde využil instanci své třídy `MailTemplateConfigHolder`, ze které jsem podle stavu účtu odpovídající název vybral. Celá metoda následně po svém dokončení vracela řetězec obsahující zpracovanou šablonu.

Díky ní jsem měl vše potřebné k tomu, abych mohl výsledný e-mail vytvořit a odeslat. Pro tento účel jsem si v návaznosti na to vytvořil další metodu, která měla za úkol celou zprávu poskládat. K tomu sloužila třída `MimeMessage` a `MimeMessageHelper`, díky nimž jsem mohl definovat příjemce zprávy, e-mailovou adresu odesilatele, předmět zprávy a její text. Takto vytvořený objekt jsem z metody vracel a následně ho využil jako argument metody `send` třídy `JavaMailSender`. Ta už se jen dále postarala o to, aby byla zpráva odeslána. Nicméně k tomu bylo ještě zapotřebí do konfigurace uvést údaje o příslušném SMTP serveru, který by tuto funkci obstaral.

Obdobným způsobem jsem doimplementoval i zbylé metody ve třídě `MailServiceImpl` a jejich volání jsem následně doplnil do patřičných metod.

5.6 Implementace rozhraní REST

Jako svůj další úkol, který jsem dostal, bylo naimplementovat rozhraní REST. To v aplikaci umožňuje přístup ke zdrojům a jejich manipulaci primárně prostřednictvím sady URI adres a HTTP metod. Díky tomu není aplikace vázaná pouze na jednu platformu, ale může být využívána celou řadou dalších aplikací nebo služeb, které podporují tvorbu HTTP požadavků. Tyto požadavky obvykle bývají složeny z několika částí, mezi něž patří cílová adresa, HTTP metoda, seznam hlaviček a parametrů nebo tělo požadavku. Z nich ovšem hrají nejdůležitější roli při mapování právě první dvě zmíněné části, a to cílová adresa a HTTP metoda. Jejich správné použití má zároveň značný vliv na přehlednost celého rozhraní, a tudíž jsem se při jejich návrhu

řídil několika pravidly a zásadami, které popisuji spolu s postupem tvorby takového rozhraní dále v této kapitole.

V naší aplikaci jsem začal implementaci REST rozhraní tvorbou tříd představující tzv. controllery, kde každý z těchto conrollerů představuje jeden zdroj, k němuž poskytuje přístup. Takovým zdrojem v aplikaci může být například nabídka, poptávka, firma nebo organizace. Pro ty každý z controllerů umožňuje jejich vytvoření, úpravy, smazání nebo načtení podle různých kritérií. Abychom mohli vytvořenou třídu považovat za REST controller, je potřeba nad ni nejprve umístit anotaci `@RestController`. Kromě ní je ale dobré zde uvést ještě jednu anotaci a to `@RequestMapping`, která umožňuje mapování metod a tříd na tzv. koncové body. To v mém případě zajistí to, že všechny metody obsažené v dané třídě budou mapovány na stejný koncový bod. Díky tomu můžeme rozlišovat jak jednotlivé zdroje a jejich metody tak i jednoduše seskupovat vícero metod pod jeden koncový bod. Například pro controller poskytující přístup k nabídkám, jsem v anotaci `@RequestMapping` definoval jako její hodnotu řetězec *„/supplies“*. Při pojmenování jsem zároveň použil množné číslo, a to z toho důvodu, že takovéto mapování označuje controller, který umožňuje práci nad všemi záznamy nabídek a nejenom jedné.

Dále jsem pokračoval tvorbou metod, které měly představovat jednotlivé koncové body pro základní CRUD operace. První takovou metodou byla metoda `create` pro vytvoření nové nabídky. Ta přijímala jako parametr objekt typu `SupplyDTO`, což bylo mimo jiné vyznačeno anotací `@RequestBody` a vracela objekt typu `ResponseEntity`. Ten představuje samotnou odpověď, která je zaslána ze serveru, a to v podobě JSON objektu. Nad tuto metodu jsem dále umístil anotaci `@PostMapping` značící to, že daná metoda bude spuštěna v reakci na požadavek odeslaný na server HTTP metodou POST s cílovou adresou odpovídající danému koncovému bodu. Ta v případě metody `create` končila řetězcem *„/supplies“*, protože jsem spolu s anotací `@PostMapping` neuvedl další část cesty, a tudíž se použila ta, která je uvedena v anotaci `@RequestMapping` nad třídou `SupplyController`. Takovéto spojení HTTP metody POST a uvedené cesty má také svůj účel, neboť vyjadřuje, že odesláním daného požadavku se provede vložení nového záznamu do kolekce všech nabídek. Zároveň jsem se používáním HTTP metod k vyjádření operací vyhnul nutnosti zapisovat do cesty slovesa, které obecně není vhodné při tvorbě koncových bodů v rozhraní REST používat. [10]

Kromě parametru `supply` jsem zde použil ještě jeden, a to typu `UriComponentsBuilder`, který mi umožnil sestavit URI cestu odkazující se na koncový bod, odkud je možné nově vytvořenou nabídku získat. Takto sestavená cesta je ve výsledku přiložena v odpovědi jako hlavička *Location* a předána klientovi. Hlavní výhodou předání takovéto odpovědi je zejména nízká náročnost na objem přenesených dat, neboť je přenesena pouze cesta obsahující identifikátor. Ten je zároveň pro klienta jedinou podstatnou informací, neboť rozdíl mezi přijatým objektem a tím, který je výsledkem zpracování, je ve většině případů pouze ve vygenerovaném identifikátoru a v auditních datech, která nejsou zapotřebí vracet. Aby ale bylo získání objektu jednodušší, je tento identifikátor umístěn již do hotové cesty, která odpovídá patřičnému koncovému bodu. Jak je vidět na ukázce ve výpisu 6, k sestavení výsledné odpovědi je již připravena třída

`ResponseEntity`, která poskytuje statickou metodu `created`. Ta slouží pro sestavení odpovědi zahrnující hlavičku *Location* a HTTP status 201, který informuje o úspěšném vytvoření.

```
@PostMapping
    public ResponseEntity create(@Valid @RequestBody final SupplyDTO supply,
        final UriComponentsBuilder uriBuilder) {
        final SupplyDTO result = supplyService.create(supply);
        UriComponents uriComponents = uriBuilder.path("supplies/{supplyId}").
            buildAndExpand(result.getId());
        return ResponseEntity.created(uriComponents.toUri()).build();
    }
```

Výpis 6: Mapování metody create

Další metodou, kterou jsem pokračoval, byla metoda `update` sloužící pro aktualizaci již existující nabídky. Její podoba je v podstatě stejná jako u metody `create`, až na to, že je nad ní umístěna anotace `@PutMapping` a vrací aktualizovaný objekt. I zde ale zůstává cesta stejná jako u předchozí metody, jelikož nám k rozlišení od ostatních mapování stačí pouze HTTP metoda. Po implementaci této metody jsem pokračoval další, a to pro smazání. U ní jsem použil anotaci `@DeleteMapping` s definovanou cestou obsahující proměnnou. Ta představovala identifikátor daného záznamu, který měl být smazán a její mapování na patřičný parametr v metodě bylo zajištěno pomocí anotace `@PathVariable`. U tohoto parametru je dále důležité, aby byl pojmenován stejně jako proměnná v cestě, na kterou se má vázat. Kromě již zmíněné anotace pro mapování, jsem nad metodu umístil ještě jednu anotaci, a to `@ResponseStatus` spolu s argumentem `HttpStatus.NO_CONTENT`, která klientovi oznamuje, že v odpovědi nebude vrácen žádný obsah.

Mezi posledními dvěma metodami, které měly poskytovat základní operace, byla i metoda pro vrácení jedné konkrétní nabídky určené jejím identifikátorem. V tomto případě jsem si vytvořil potřebnou metodu, nad níž jsem umístil anotaci `@GetMapping`, zajišťující potřebné mapování. Jako součást tohoto mapování jsem následně definoval proměnnou, kterou jsem svázal s parametrem metody pomocí anotace `@PathVariable`. V těle metody jsem tento identifikátor dále využil pro získání objektu třídy `Optional`, pomocí jehož metod jsem vytvořil i výsledný objekt třídy `ResponseEntity`. Díky němu jsem mohl spolu s odpovědí vrátit také HTTP status, který značil, zda byla nabídka nalezena či nikoliv. Výslednou podobu metody je možné vidět ve výpisu 7.

```
@GetMapping("/{supplyId}")
    public ResponseEntity<SupplyDTO> getOneById(@PathVariable final UUID
        supplyId) {
        return supplyService.getOneById(supplyId)
```

```
            .map(result -> new ResponseEntity<>(result, HttpStatus.OK))
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }
```

Výpis 7: Mapování metody getOneById

Druhá z metod dále sloužila k vrácení paginovaného seznamu nabídek dle předaných parametrů v adrese požadavku. Názvy a formát těchto parametrů jsou určeny frameworkem Spring Boot, který provádí jejich mapování na objekt typu `Pageable`. Mezi tyto parametry patří zejména parametr `page`, který určuje číslo stránky nebo parametr `size`, který určuje její velikost. Kromě nich lze ještě definovat parametr `sort`, který určuje, podle jakých atributů se budou výsledky řadit. Takto sestavený objekt je navíc kompatibilní s frameworkem Spring Data, který ho dokáže využít při tvorbě odpovídajícího dotazu, u něhož je předpokládáno právě využití paginace. Ukázka adresy požadavku v níž jsou některé parametry použity, vypadá následovně:

```
http://localhost:6080/supplies?page=0&size=10
```

Stránky jsou zde počítány od nuly, přičemž s každou odpovědí se vrací i informace o tom, jaký je celkový počet dostupných stránek a kolik záznamů je možné celkem získat.

5.7 Validace

Validace v našem projektu byla pokryta balíčkem *javax.validation*, která k tomuto účelu nabízela jednak bohaté rozhraní, ale také sadu anotací, skrze něž bylo možné definovat řadu pravidel a omezení, které bylo nutné u vstupních objektů dodržovat. Mezi těmito anotacemi byly i ty, které se vážou k textovým atributům. Z nich jsem použil například anotaci `@NotBlank`, která značí, že daný atribut nemůže nabývat hodnoty *null* a jeho řetězcová reprezentace musí obsahovat alespoň jeden znak, který není považován za tzv. bílý znak, což může být mezera nebo odřádkování. Kromě ní jsem z této kategorie využil i anotaci `@Email`, která se starala o to, aby daný atribut, nad nímž je umístěna, obsahoval správně utvořenou e-mailovou adresu. To se mi následně hodilo při registraci, kdy jedním z údajů, které uživatel musel uvést, byla právě e-mailová adresa, na níž pak byl zaslán e-mail vyzývající k potvrzení registrace. S registrací ovšem přicházely i další řetězcová omezení, k nimž také patřila pevně daná délka atributu pro uvedení hodnoty IČO. Ta v našem případě musela mít délku 8 znaků, což ostatně bylo dáno anotací `@Size`, kterou jsem nad tento atribut umístil. Do ní jsem pro tento účel uvedl hodnoty dvou parametrů, a to `min` a `max`, které měly v tomto případě totožnou hodnotu a to 8.[11]

Mezi požadavky na validaci se našly ale i takové, které s textem nepracují. Jedním z nich byla také kontrola rozsahu číselného hodnocení. To mohlo v našem případě nabývat hodnot 1 až 5, což jsem zajistil pomocí dvou anotací `@Min` a `@Max`, které určovaly minimální a maximální možnou hodnotu. Druhou obdobnou anotací, kterou jsem často využíval, byla anotace `@NotNull`, která zajišťovala to, aby daný atribut neobsahoval referenci na *null*. K ní dále ještě existoval opak, a to v podobě anotace `@Null`, která říkala, že daný atribut musí referenci na *null* obsahovat. Ta

se hodila v případě vynucení toho, aby některé z atributů nebyly zasílány. Příklad mého použití je vidět ve výpisu 8.[11]

```
public class CompanyDTO {
    private UUID accountId;
    @NotBlank
    @Size(min = 8, max = 8)
    private String ico;
...
}
```

Výpis 8: Použití anotací pro validaci

Samotná validace následně v aplikaci fungovala plně automaticky. Jediné, co k tomu bylo potřeba, bylo správné označení parametrů metod, při jejichž volání mělo k validaci dojit. Pro tento účel jsem využil anotaci `@Valid`, kterou jsem umístil k parametrům metod, nad nimiž bylo provedeno mapování na jednotlivé koncové body. Tím jsem tak dosáhnul toho, že u objektu, který měl být skrze rozhraní REST uložen, bude nejdříve před předáním do patřičné metody v controlleru provedena validace. Díky tomu jsem mohl posléze volaným službám předat správně sestavený objekt k dalšímu zpracování.

6 Nasazení projektu do produkčního prostředí

Posledním mým úkolem na projektu bylo jeho nasazení do produkčního prostředí, které mělo být v našem případě tvořeno cloudovými službami Amazon Web Services (AWS). Průběh nasazení do takovéhoto prostředí se dělil na několik částí, do nichž spadala řada úkonů spojených s hledáním vhodných služeb, s jejich konfigurací a propojením a v neposlední řadě také s výsledným nasazením a spuštěním celého systému.

6.1 Cloudové prostředí AWS

Cloudové služby AWS jsme si zvolili z několika důvodů, mezi nimiž dominovala jednoduchá správa všech prostředků pomocí webového rozhraní, okamžitá dostupnost služeb včetně požadované výpočetní kapacity a úložiště nebo přijatelná cena. Díky tomu jsme si nemuseli dělat starosti s obstaráváním potřebných hardwarových prostředků, s jejich údržbou a se škálováním, což nám ušetřilo spoustu času, který jsme mohli věnovat vývoji a odladění celého projektu. Kromě toho nám tyto služby přinesly i řadu již předpřipravených řešení v podobě nakonfigurovaného databázového serveru nebo sdíleného veřejně dostupného úložiště.

6.2 Volba služeb AWS

Prvním krokem, který bylo pro běh projektu nutné učinit, byla volba cloudových služeb. Ty měly za cíl pokrýt veškeré nároky týkající se technologií i nefunkčních požadavků. Mezi nimi jsem proto musel najít odpovídající služby pro provoz námi využitých databázových technologií, aplikačního serveru s podporou Javy a NodeJS nebo pro provoz vyhledávacího enginu Elasticsearch. K tomu naštěstí byla k dispozici řada služeb. Například pro relační databázi jsem si zvolil službu Amazon RDS, která poskytovala jednoduchou konfiguraci s pokročilými možnostmi škálování, zálohování, řízení přístupu a monitoringu. Díky této službě jsem nemusel ručně vytvářet a konfigurovat samostatnou instanci pro provoz databáze nebo se starat o zálohy. Pro aplikační servery jsem poté využil služby Amazon EC2, která poskytovala možnosti správy výpočetní kapacity. V rámci ní jsm si mohl vytvořit virtuální instanci s požadovanými parametry, na kterou jsem nakonec mohl aplikaci nasadit. Kromě ní jsem dále využil ještě služby Amazon S3, která mi poskytla objektové úložiště pro nejrůznější data, od statických souborů celé webové aplikace až po ukládání veškerých obrázků, a to i těch uživatelských. Díky tomu jsem proto nemusel řešit souborový server, jeho konfiguraci, kapacitu a ani škálování. Vše totiž bylo již v rámci služby zajištěno a připraveno k použití. Co se dále týče propojení všech těchto služeb po síťové stránce, tak to bylo zajištěno díky tzv. virtuálnímu privátním cloudu – Amazon Virtual Private Cloud. Díky němu bylo možné si vytvářet svoje vlastní virtuální oddělené sítě, spravovat jejich adresní prostor, upravovat směrovací tabulky nebo vytvářet další podsítě. Mimo to bylo možné v rámci takovéto sítě definovat i bezpečnostní skupiny a tzv. ACL listy.

Nicméně ne vždy se našla taková služba, která by splňovala všechny naše požadavky. Takovým požadavkem bylo i využití dokumentové databáze MongoDB, která v katalogu chyběla. Místo ní ovšem Amazon v rámci AWS nabízel svou alternativu a to DynamoDB, která fungovala sice na obdobném principu jako MongoDB, avšak její použití by pro nás znamenalo značné úpravy v rámci celého projektu. Proto jsem se rozhodl v tomto případě využít jako databázový server instanci EC2, která se k tomuto účelu perfektně hodila. Nicméně její použití již vyžadovalo ruční instalaci a následnou konfiguraci, která ale v případě databáze MongoDB nebyla nikterak složitá.

Obdobná situace mě nicméně čekala i u zprovoznění vyhledávacího enginu Elasticsearch. Pro něj sice existovala služba jménem AWS Elasticsearch Service, avšak po jejím počátečním spuštění a konfiguraci, jsem přišel na to, že takto spuštěná instance Elasticsearch neumožňuje připojení klientů skrze TCP spojení na portu 9300, které naše aplikace vyžadovala. Místo toho zde bylo dostupné pouze spojení pomocí REST rozhraní běžícího na portu 9200, což ovšem nebylo možné v našem případě použít. Proto i zde jsem se rozhodl využít instanci EC2, na kterou jsem Elasticsearch nainstaloval a nakonfiguroval tak, aby jej bylo možné v rámci našeho projektu použít.

Kromě základních služeb bylo součástí portfolia AWS i několik pokročilejších, které měly za cíl usnadnit činnost například spojenou s nasazením webové aplikace. Takovéto služby jsem se za účelem ulehčení práce pokusil využít i já. Nicméně jsem se v průběhu jejich používání přesvědčil o tom, že tyto služby mohou kromě výhod přinášet i řadu nevýhod v podobě nejrůznějších problémů. Na ně jsem ostatně narazil i u služby AWS Elastic Beanstalk, která mi měla značně ulehčit celý proces nasazení. Bohužel jsem se místo ulehčení musel potýkat s celou řadou komplikací, které ve výsledku celý proces značně prodloužily. Hlavním problémem u této služby pro mě představovala práce s logováním a laděním. To bylo stejně jako vše řešeno přes webové rozhraní, přes nějž bylo možné si logy stáhnout, avšak pouze za předpokladu, že aplikace se kompletně celá spustila. Díky tomu bylo velice obtížné zjistit, proč aplikace spadla již při startu, neboť po pádu vždy docházelo k restartování celé instance. To mělo poté za následek i několikaminutové prodlevy, během nichž nebylo možné tyto logy získat a chyby tak opravit. Z tohoto důvodu jsem se nakonec rozhodl si vše udělat ručně pomocí instancí EC2, které jsem ostatně již využíval pro několik dalších služeb.

6.3 Konfigurace služeb

Po ujasnění všech použitých služeb jsem se pustil do jejich konfigurace. Tu jsem začal u služby Amazon RDS umožňující správu relačních databází. Pro tento účel jsem využil její webové rozhraní a skrze nabízeného průvodce jsem si během několika kroků založil novou databázi. Během nich jsem postupně definoval použitý engine, což v našem případě byla databáze PostgreSQL, účel použití, jímž bylo produkční využití, její název a přístupové údaje. Kromě toho zde bylo nutné také zvolit parametry instance, ve které to všechno poběží, což pro naše účely postačovala instance typu *db.t2.micro*. Ta disponuje jedním virtuálním procesorovým jádrem a 1 GB

operační paměti. Jako úložiště jsem zde zvolil SSD s kapacitou 20 GB a typem *General Purpose (SSD)*. Při této příležitosti jsem nakonec ještě nastavil automatické zálohování, pro nějž jsem vybral časový interval, ve kterém se bude provádět a v neposlední řadě také intenzitu.

Po založení se následně nově vytvořená instance objevila v seznamu spravovaných databází a bylo možné se podívat na její detaily. Mezi nimi kromě základních informací byl zobrazen i přehled o jejím využití, počtu připojení nebo dostupném volném úložišti. Kromě toho bylo možné z této stránky provádět i údržbu nebo si prohlížet záznamy z logu. Ze zbytku funkcionality jsem zde ale nejvíce oceňoval možnost vytvářet zálohy. Ty byly řešeny formou tzv. *Snapshots*, které obsahovaly kompletní obraz databáze a jejich dat. Vyjma toho se zde nacházel ještě jeden podstatný údaj, a to adresa serveru, přes kterou se bylo možné k databázi připojit. Pak už jen stačilo v databázi vytvořit potřebná schémata a naplnit je tabulkami.

Další službou, kterou jsem pokračoval, byla služba Amazon EC2. Ta byla pro náš projekt zároveň tou nejdůležitější, neboť zajišťovala veškerou výpočetní kapacitu potřebnou pro běh backendové a frontendové částí a pro zbytek služeb, které nebyly obsaženy v katalogu AWS. Těmi jsem nakonec i začal a stejným způsobem jako u relační databáze, jsem skrze průvodce postupně vytvářel potřebné instance.

První takovou instancí byla instance zajišťující běh dokumentové databáze MongoDB. Pro ni bylo v prvním kroku nutné vybrat tzv. Amazon Machine Image (AMI), který sloužil jako šablona obsahující potřebnou konfiguraci včetně operačního systému, aplikačního serveru nebo jiných služeb. K tomu zde bylo na výběr několik desítek různých obrazů, které se lišily jak operačním systémem, tak i svým účelem použití. Já si ovšem ve všech případech vždy vybral obraz Amazon Linux 2 AMI, který v základu obsahoval vše potřebné a navíc byl optimalizovaný pro běh na instancích EC2. K němu bylo dále nutné vybrat typ instance podle předpokládaného využití, přičemž hlavními rozdíly mezi nimi tvořilo počet virtuálních procesorových jader, množství operační paměti, typ úložiště a síťová kapacita. Pro účel provozu databáze MongoDB jsem si v tomto případě zvolil ten nejzákladnější typ instance a to *t2.nano*. Ta disponuje jedním virtuálním procesorovým jádrem a 512 MB operační paměti, což pro náš nenáročný provoz bohatě dostačuje.

V dalším kroku se dále nastavovaly detaily použité instance. Mezi nimi bylo možné si zvolit síť VPC, podsíť v závislosti na zóně dostupnosti, rezervaci kapacity, monitoring nebo třeba detaily ohledně toho, zda se má jednat o dedikovanou instanci nebo o instanci sdílející svůj hardware s ostatními. Zde ale bylo nutné mít na paměti, že zrovna tři poslední jmenované možnosti navyšují i výslednou cenu, a proto bylo nutné volit tyto nastavení s rozvahou. Kromě toho se zde ještě nacházelo několik možností, jimiž se dalo určit co se má stát po vypnutí instance nebo při pokusu o její zrušení. Díky tomu se dalo zabránit nechtěnému zrušení, které by mohlo mít fatální následky.

Čtvrtý krok dále sloužil k výběru úložiště. Zde bylo jednak možné upravit velikost a typ základního úložiště, na kterém byla instance uložena, ale také bylo možné přidávat nové svazky. Toho jsem využil i v případě instance pro MongoDB, kde jsem přidal ještě jeden svazek typu

General Purpose SSD (gp2) o kapacitě 8 GB, který měl sloužit čistě pro ukládání databázových dat. Takovéto rozdělení následně představovalo i několik výhod v podobě méně objemných záloh nebo možnosti svazek přepojit do jiné instance, a to v případě problémů či změně jejího typu.

K instanci bylo také možné připojit několik značek, které byly představovány záznamy typu *klíč-hodnota*. Pomocí nich nebyl problém k instanci doplnit užitečná metadata například v podobě typu prostředí nebo názvu instance. Této možnosti jsem využil a pro všechny instance tvořící produkční prostředí jsem přidal značku s klíčem *Environment* a hodnotou *Production*. Díky tomu jsem si později mohl pomocí filtrů zobrazit instance vážící se k určitému prostředí, se kterým jsem aktuálně potřeboval pracovat. V předposledním kroku jsem dále již jen definoval konfiguraci bezpečnostních skupin. Ty představovaly množinu pravidel, pomocí nichž se řídil síťový provoz. Díky nim bylo například možné omezit veškerý provoz, který směřoval k instanci pouze na SSH spojení navázané z určité IP adresy. Takové pravidlo jsem ostatně pro svou instanci definoval i já, a to z toho důvodu, aby se předešlo případným pokusům o neoprávněný přístup k dané instanci.

Poslední krok už jen obsahoval shrnutí všech dosavadních nastavení a umožňoval instanci spustit. Předtím bylo ale nutné si ještě nechat vygenerovat privátní klíč, který sloužil k připojení k terminálu instance skrze SSH spojení, a vše potvrdit. Po úspěšném vytvoření a spuštění se následně instance objevila na seznamu spravovaných instancí, kde bylo opět možné vidět jejich detaily. Z nich byly nejpodstatnější položky tykající se bezpečnostních skupin a IP adres. Ty byly v mém případě obvykle dvě. První sloužila jako privátní adresa přidělená v rámci zvolené podsítě a druhá jako veřejná adresa sloužící pro vzdálené připojení. Kromě nich tady byl vidět ještě přehled využití prostředků instance a kontrola stavu.

Obdobným způsobem jsem dále založil i zbytek instancí, které se už oproti výše popsané instanci pro MongoDB lišily jenom v několika parametrech. Mezi nimi byl například i typ instance, který jsem v následujících případech musel volit podle nároků jednotlivých aplikací. Pro Elasticsearch a backendovou část jsem proto musel zvolit již typ *t2.small*, který zahrnuje větší množství operační paměti a to konkrétně 2 GB. Stejný případ poté nastal i u poslední instance sloužící pro provoz frontendu, kde jsem zvolil typ *t2.micro*, který zahrnuje pouze 1 GB operační paměti. Dalším parametrem v tomto ohledu bylo dále úložiště, kde už jsem si v instancích zaměřených na backend a frontend vystačil pouze se systémovým diskem. Výjimkou byla pouze instance s Elasicsearch, kde jsem rovněž jako u instance s MongoDB, přidal dodatečný oddíl na data. A konečně posledním parametrem, u kterého nastala změna, byla veřejná IP adresa instance, na které běžel frontend. Ta totiž musela být statická a neměnná, a to z toho důvodu, aby v případě zastavení a znovu spuštění instance, nebylo nutné upravovat DNS záznam, který je na ni mapovaný. K tomu v rámci AWS sloužila tzv. *Elastic IP*, která umožňovala získat veřejnou adresu a tu následně přiřazovat podle potřeby právě k jednotlivým instancím.

6.4 Příprava EC2 instance pro backend

Dalším mým krokem byla příprava nově vytvořené EC2 instance určené pro provoz backendu. V ní bylo v rámci příprav nejprve nutné nainstalovat a nakonfigurovat běhové prostředí Java JRE 8 a vytvořit základní adresářovou strukturu pro budoucí nasazené aplikace. K tomu jsem se rozhodl využít distribuci Amazon Corretto 8, která je poskytována jako dodatečný balíček v rámci EC2 instancí založených na Amazon Linux 2. Její hlavní výhodou v tomto ohledu je dlouhodobá podpora, která zahrnuje jak bezpečnostní záplaty, tak i výkonnostní vylepšení. Kromě toho je tato distribuce založená na Open Java Development Kit (OpenJDK) a je u ní zajištěna kompatibilita s Java SE standardem, což z ní dělalo ideálního kandidáta pro naše účely. [12] [13]

Instalace této distribuce probíhala skrze repozitář dodatečných balíčků. Ten musel být ale nejdříve povolen, což se dalo zajistit příkazem:

```
sudo amazon-linux-extras enable corretto8
```

Poté bylo možné danou distribuci nainstalovat, a to skrze správce balíčků yum. Příkaz, který jsem k tomuto účelu využil, vypadal následovně:

```
sudo yum install java-1.8.0-amazon-corretto
```

Po dokončení instalace jsem si následně přítomnost správné verze ověřil příkazem `java -version` a pustil se do vytváření adresářové struktury. Ta byla v mém případě velice jednoduchá a v podstatě se jednalo o vytvoření dvou adresářů odpovídající dvěma službám, které jsem plánoval do nich umístit. Těmito službami byly v mém případě mikro služby *auth-service* a *offering-service*, které tvořily backendovou část. Jejich adresáře jsem nakonec vytvořil ve složce */var* a pustil se do sestavení produkční verze celého projektu. To se odehrávalo již v mém lokálním prostředí, kde jsem si pomocí nástroje Maven nechal produkční verzi projektu sestavit. Ještě předtím jsem ale potřeboval provést pár úprav v konfiguračním souboru *pom.xml* umístěném v hlavním adresáři projektu. V něm bylo nutné do sekce, kde byl uveden sestavovací plugin, doplnit jeden konfigurační parametr, a to `executable` s hodnotou *true*, který specifikoval to, že výsledně sestavená aplikace má být samostatně spustitelná. Díky tomu jsem následně nemusel aplikaci spouštět skrze nástroj `java` s parametrem `-jar` a mohl jsem ji tak přímo využít při tvorbě systemd jednotek. Kromě toho zde bylo nutné ještě upravit konfiguraci profilu, který měl v tomto případě odpovídat produkčnímu nastavení. V něm byly například nastaveny potřebné údaje pro připojení k relační a dokumentové databázi umístěné v cloudu, bez kterých by nešlo aplikaci korektně spustit.

K samotnému sestavení jsem potřeboval dva příkazy, přičemž prvním příkazem jsem zajistil vyčištění výstupní složky a následnou kompilaci celého projektu. Jeho podoba vypadala následovně:

```
mvn clean install
```

Druhým příkazem jsem dále zajistil to, aby došlo k zabalení aplikace do distribuovatelného formátu, jakým byl i formát JAR. Podoba celého příkazu v tomto případě byla:

```
mvn package
```

Oba uvedené příkazy jsem vždy spouštěl v kořenové složce projektu, přičemž výstupní spustitelné soubory bylo možné nalézt ve složkách *target* umístěných v každém z modulů. Ty už mi jen spolu s konfiguračními soubory zbývalo přenést do cílových adresářů, vytvořených v předchozím kroku. K tomu jsem využil program WinSCP, který mi umožnil se k instanci připojit skrze SSH a sobory nahrát.

Po tomto kroku jsem se dále vrátil zpět do terminálu a pokračoval v definici služeb. K nim ovšem bylo nutné ještě vytvořit dva uživatelské účty, pod kterými by běžely. K tomu jsem využil příkaz `useradd` s parametrem `-s`, který pro nový účet přiřazoval zadaný terminál. V mém případě jsem pro tyto účty žádný terminál nepotřeboval a proto jsem spolu s ním uvedl cestu */sbin/nologin*. Podoba celého příkazu pro vytvoření účtu ke službě *offering-service* vypadala následovně:

```
sudo useradd -s /sbin/nologin offering-service
```

Kromě vytvoření účtů, bylo nutné přiřadit k nakopírovaným souborům i vlastníka a upravit oprávnění, neboť bez něj by jinak nebylo možné službu pod vybraným uživatelem spustit. Pro tento případ jsem pomocí příkazu `chown` přiřadil k souborům dané služby potřebného vlastníka a pomocí příkazu `chmod` upravil jejich oprávnění. To v případě spustitelného souboru zahrnovalo bitovou masku *500*, která povolovala vlastníkovi soubor spustit a číst z něj. U konfiguračních souborů mi následně postačila bitová maska *400*, která opravňovala vlastníka souboru pouze k jeho čtení. Ukázka použití příkazu `chmod` pro úpravu oprávnění vážící se ke konfiguračnímu souboru *application-prod.yml* vypadala následovně:

```
sudo chmod 400 application-prod.yml
```

Po dokončení všech úprav v adresářích jsem se mohl pustit do vytváření systemd jednotek, které sloužili k definování potřebných parametrů služeb. Začal jsem nejprve tím, že jsem si vytvořil v adresáři */etc/systemd/system* dva soubory *auth-service.service* a *offering-service.service*. Ty představovaly jednotlivé služby a jejich obsahem byla řada parametrů určujících popis jednotky, kdy se má služba spustit, pod jakým uživatelem se má spustit, a nakonec také jaký příkaz se má při jejím spuštění vykonat. Kromě toho zde byl ještě uveden status při úspěšném ukončení a detaily instalace jednotky.

Po úspěšném vytvoření konfiguračních souborů jsem ještě zajistil jejich automatické spuštění po startu, a to pomocí příkazu `systemctl enable`. Jeho podoba v případě služby *offering-service* vypadala následnovně:

```
sudo systemctl enable offering-service.service
```

Pak již jen stačilo buď instanci restartovat nebo jednotlivé služby spustit ručně. Já se rozhodl pro poslední jmenovanou možnost a pomocí příkazu `systemctl start` jsem každou ze služeb ručně spustil. O úspěšném spuštění jsem se poté mohl přesvědčit skrze příkaz `systemctl status`, jehož výstupem byla informace o tom, v jakém stavu se služba nachází.

Kromě stavu bylo nicméně možné získat i podrobnější informace o běhu aplikace, a to pomocí žurnálu. Ten byl přístupný skrze příkaz `journalctl` a obsahoval kompletní log služby, ve kterém byl vidět průběh startu webové aplikace nebo případné chyby, které během jejího provozu nastaly. Podoba daného příkazu pro výpis logu služby *offering-service*, který jsem často využíval, vypadala následovně:

```
journalctl -u offering-service.service -f
```

Po úspěšném spuštění obou služeb jsem se poté pustil do přípravy instance určené pro provoz frontendu.

6.5 Příprava EC2 instance pro frontend

Příprava instance pro běh frontendu probíhala obdobně jako u backendu. Vše opět začalo u instalace běhového prostředí, které v tomto případě bylo představováno aplikací NodeJS. K tomu jsem využil aplikaci Node Version Management (NVM), která mimo jiné umožňovala instalaci několika verzí zároveň. Její stažení probíhalo skrze příkaz `curl`, přičemž jeho použití vypadalo následovně:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh |
bash
```

Po stažení poté bylo nutné aplikaci NVM aktivovat, k čemuž sloužil příkaz:

```
. /.nvm/nvm.sh
```

K použití, a tedy i následné instalaci jsem dále použil příkaz `nvm install` za nímž jsem specifikoval verzi aplikace NodeJS. Tu jsem v našem případě zvolil 10.14.2, což byla také v té době nejnovější verze, jaká byla dostupná. Její kompatibilitu jsem následně ještě ověřil ve svém lokálním prostředí, tak aby byl později zajištěn v tomto ohledu bezchybný provoz. [14]

Kromě běhového prostředí jsem do instance doinstaloval i službu Nginx. Tu jsem se rozhodl použít pro sjednocení veškeré komunikace na jeden server, a to z toho důvodu, abych mohl jednoduše zajistit šifrované spojení skrze SSL. Dalším důvodem k tomuto řešení bylo mimo jiné i to, že nabízená alternativní služba Amazon API Gateway vyžadovala pro splnění našich požadavků provoz služby Elastic Load Balancing. Ta sice umožňovala nasazení SSL certifikátu, který bylo možné získat zdarma skrze službu AWS Certificate Manager, nicméně jeho použití značně zvyšovalo náklady na provoz. Z tohoto důvodu jsem proto nakonec využil funkce reverzní

proxy v aplikaci Nginx, a pro získání SSL certifikátu poté služeb jedné z certifikačních autorit, která nám jej vystavila.

Instalace služby Nginx opět vyžadovala doplňkový repozitář *amazon-linux-extras*, který byl součástí mé distribuce a po jejím dokončení jsem se pustil do konfigurace. Ta v mém případě musela být dvojího typu, a to z toho důvodu, že jsem konfiguraci a spuštění serveru prováděl ještě před oficiálním spuštěním webu. Kvůli tomu musela být připravena ještě jedna stránka, která informovala návštěvníky o připravovaném portálu a datu jeho uvedení do provozu. K tomu jsem využil přímo možnosti služby nginx, která funkci webového serveru poskytovala. Díky tomu mi stačilo pouze do instance nahrát potřebné stránky a službu spustit. Mimo to ale bylo potřeba provést ještě patřičnou úpravu DNS záznamu pro doménu, přes kterou měly být stránky veřejně dostupné. Pro tento účel jsem využil IP adresu přiřazenou skrze službu Elastic IP a vložil ji do DNS záznamu typu *A*. Poté stačilo již vyčkat, než se změna projeví a stránky tak budou dostupné i přes běžnou adresu.

Kromě konfigurace pro dočasné statické stránky jsem později potřeboval provést nastavení i pro ty, které již představovaly hotovou webovou aplikaci. V rámci ní jsem nastavil všechny potřebné cesty a parametry nutné pro správný chod reverzní proxy a SSL připojení. Kromě toho jsem při té příležitosti nainstaloval i potřebný certifikát a celou konfiguraci následně ověřil příkazem, který provedl její kontrolu. Jeho podoba vypadala následovně:

```
nginx -t
```

Po úspěšné kontrole jsem ale službu ještě nespouštěl a místo toho jsem se pustil do sestavení produkční verze frontendu. Ta vyžadovala využití nástroje Node Package Manager (NPM), který při sestavení mimo jiné zajišťoval i minifikaci, obfuskaci a optimalizaci kódu, tak aby výsledné soubory byly co nejmenší a aby byl zajištěn plynulý chod aplikace. Takto vygenerované soubory jsem následně přesunul jednak na server do instance EC2 určené pro frontend, ale jejich část také do služby Amazon S3, která umožňovala hostování statického obsahu. Tato duplikace byla zapotřebí hlavně z důvodu použití technologie Server Side Rendering (SSR), která umožňovala vykreslování stránek na serveru za účelem jednak rychlejšího načítání ale i kvůli podpoře při indexování stránky ve vyhledávačích. Tato technologie totiž zajišťovala to, že při prvotním načtení stránky je do prohlížeče zaslána již vykreslená statická stránka, přičemž načítání poměrně objemného javascriptového balíčku obsahující kód aplikace je prováděno na pozadí. Díky tomu je následně zajištěna i podpora při indexování stránky vyhledávači, neboť ty obvykle JavaScript nepodporují a řídí se pouze statickým obsahem. Kromě toho tato technologie nalezla uplatnění i při sdílení obsahu na sociálních sítích, neboť ty ke svému účelu rovněž vyžadovaly statickou stránku. [15]

Jakmile jsem všechny potřebné soubory nakopíroval na správná místa, mohl jsem se přesunout do terminálu na serveru a aplikaci spustit. K tomu mi posloužil příkaz `node`, spolu s nímž jsem uvedl zdrojový soubor celé aplikace. Jeho podoba poté vypadala takto:

```
node server.js
```

K úspěšnému zpřístupnění aplikace ale bylo nutné ještě upravit nastavení služby Nginx tak, aby po připojení na server skrze protokol HTTP, případně HTTPS se již nezobrazovala dočasná stránka, ale ta, která představovala hotovou aplikaci. Po dokončení těchto úprav již jen stačilo stránky znovu navštívit a zkontrolovat jejich funkčnost.

7 Uplatněné a chybějící znalosti

7.1 Znalosti získané na vysoké škole a uplatněné v praxi

Znalostí získaných během studia na vysoké škole a následně uplatněné v praxi byla v mém případě celá řada. Mezi ty nejpodstatnější bych si zde dovolil především zmínit ty, které se týkaly programování, algoritmizace a obecně softwarového inženýrství, protože jejich využití během celého vývoje projektu hrálo nejdůležitější roli. Jako jeden z příkladů bych zde rád zmínil znalosti nabyté v předmětu Vývoj informačních systémů, které mi poskytly jednak základní náhled na postupy využívané při vývoji informačních systémů, ale také řadu znalostí z oblasti návrhových vzorů. Kromě tohoto předmětu mi značně pomohl i předmět Java technologie, který mi poskytl dobré základy a přehled ve vývoji enterprise aplikací založených na platformě Java. Ty se například hodily při vývoji perzistentní vrstvy nebo při obecné práci s webovými technologiemi postavenými na Java platformě.

Důležitou roli hrály ale i znalosti získané z dalších předmětů, a to především z těch, které byly zaměřeny na návrh a využití databázových systémů. Například při návrhu databázového modelu a tvorbě nativních dotazů jsem využil znalosti získané z předmětu Úvod do databázových systémů a při práci s databázovými transakcemi jsem zase využil znalosti získané během výuky předmětu Databázové a informační systémy.

Z dalších znalostí, bych si ještě dovolil zmínit ty, které se týkaly počítačových sítí a správy operačních systémů. Těch jsem během své praxe využil při práci na posledních úlohách, které se týkaly nasazení projektu do cloudového produkčního prostředí a při nichž jsem čerpal i ze znalostí získaných během výuky předmětů Počítačové sítě a Správa operačních systémů.

7.2 Chybějící znalosti

Mezi chybějící znalosti bych naopak zařadil ty, které se týkaly práce s jinými typy databází jako byla v mém případě dokumentová databáze MongoDB. Její využití pro nás sice v konečném důsledku představovalo řadu výhod, avšak všechny informace spojené s jejím využitím na projektu jsem si musel nastudovat sám. Kromě těchto znalostí jsem ale postrádal i ty, které se týkaly využití vyhledávacího enginu Elasticsearch. Ten jsem se na projektu rozhodl využít především z toho důvodu, že poskytuje jednak mnohem větší možnosti vyhledávání než při použití relační databáze a dotazovacího jazyka SQL, ale také mnohem vyšší rychlost při vyhledávání i napříč velkým objemem dat. Jeho využití se ovšem neobešlo bez dodatečného nastudování nejenom jeho využití, ale i způsobu nasazení do produkčního prostředí.

8 Závěr

V rámci své odborné praxe jsem měl možnost pracovat v týmu zkušených vývojářů, a to na projektu s reálným využitím. Během ní jsem získal řadu zkušeností nejenom s návrhem architektury, ale také s následným vývojem, při němž jsem se naučil pracovat s řadou moderních technologií. Mezi nimi mimo jiné nechyběl framework Spring Boot, ORM framework Hibernate, dokumentová databáze MongoDB anebo vyhledávací engine Elasticsearch. Kromě toho jsem se naučil pracovat i s populárním cloudovým prostředím AWS a jeho službami, jenž jsem později využil při nasazení celého projektu do produkčního prostředí. Mimo to jsem měl dále možnost setkat se s agilním vývojem za využití metodiky Scrum nebo si vyzkoušet práci v týmu.

Ve výsledku nakonec vznikl nový webový portál, jenž by měl v budoucnu sloužit k navazování spoluprací mezi organizacemi a společnostmi z neziskového, veřejného a soukromého sektoru. Své uplatnění najde zejména při podpoře společensky prospěšných aktivit a dalšímu prohlubování spolupráce mezi uvedenými subjekty.

Závěrem bych své absolvování odborné praxe hodnotil velmi kladně a jsem rád za to, že mi takováto možnost byla poskytnuta, neboť jsem při ní nejenomže získal cenné zkušenosti, které se budou hodit v budoucím zaměstnání nebo na vysoké škole, ale také jsem napomohl dobré věci.

Literatura

[1] Our company | Tieto. *A leading Nordic IT services and software company | Tieto* [online]. Espoo: Tieto, c2019 [cit. 2019-03-24]. Dostupné z: https://www.tieto.com/cz/about-us/our-company/

[2] Odvětví | Tieto - Kariéra. *Navrhni si vlastní pozici - Tieto* [online]. Ostrava: Tieto, c2019 [cit. 2019-03-24]. Dostupné z: https://jobs.tieto.cz/co-delame/odvetvi/

[3] Our company | Tieto. *A leading Nordic IT services and software company | Tieto* [online]. Espoo: Tieto, c2019 [cit. 2019-03-24]. Dostupné z: https://www.tieto.com/en/about-us/our-company/

[4] Hibernate ORM 5.4.1.Final User Guide. *Dashboard - JBoss.org Documentation* [online]. Raleigh: Red Hat, 2019 [cit. 2019-03-18]. Dostupné z: http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#identifiers-generators-uuid

[5] Mapping Enums Done Right With @Convert in JPA 2.1. *DZone: Programming & DevOps news, tutorials & tools* [online]. Morrisville: Dzone, 2013 [cit. 2019-03-18]. Dostupné z: https://dzone.com/articles/mapping-enums-done-right

[6] The best way to fix the Hibernate "HHH000104: firstResult/maxResults specified with collection fetch; applying in memory!" warning message. *Vlad Mihalcea* [online]. Cluj-Napoca: Mihalcea, 2019 [cit. 2019-03-18]. Dostupné z: https://vladmihalcea.com/fix-hibernate-hhh000104-entity-fetch-pagination-warning-message/

[7] Hibernate Tips: How to fetch associations in batches. *Thoughts on Java* [online]. Paderborn: Janssen, c2019 [cit. 2019-03-18]. Dostupné z: https://thoughts-on-java.org/hibernate-tips-how-to-fetch-associations-in-batches/

[8] JPA 2.2 Introduces @Repeatable Annotations. *Thoughts on Java* [online]. Paderborn: Janssen, c2019 [cit. 2019-03-24]. Dostupné z: https://thoughts-on-java.org/jpa-2-2-repeatable-annotations/

[9] Tutorial: Using Thymeleaf. *Thymeleaf* [online]. The Thymeleaf Team, 2018 [cit. 2019-03-24]. Dostupné z: https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#text-literals

[10] RESTful API Design-Step By Step Guide. *Hacker Noon* [online]. Colorado: Artmap, 2018 [cit. 2019-03-24]. Dostupné z: https://hackernoon.com/restful-api-design-step-by-step-guide-2f2c9f9fcdbf

[11] Java Bean Validation Basics. Baeldung | *Java, Spring and Web Development tutorials* [online]. Bukurešť: Baeldung, 2018 [cit. 2019-03-24]. Dostupné z: https://www.baeldung.com/javax-validation

[12] Amazon Corretto. Amazon Web Services (AWS) - *Cloud Computing Services* [online]. Seattle: Amazon Web Services, c2019 [cit. 2019-03-24]. Dostupné z: https://aws.amazon.com/corretto/

[13] Amazon Corretto 8 Installation Instructions for Amazon Linux 2. *AWS Documentation* [online]. Seattle: Amazon Web Services, c2019 [cit. 2019-03-24]. Dostupné z: https://docs.aws.amazon.com/corretto/latest/corretto-8-ug/amazon-linux-install.html

[14] Tutorial: Setting Up Node.js on an Amazon EC2 Instance. *AWS Documentation* [online]. Seattle: Amazon Web Services, c2019 [cit. 2019-03-24]. Dostupné z: https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/setting-up-node-on-ec2-instance.html

[15] Server Side Rendering with React. *Flavio Copes* [online]. Copes, 2018 [cit. 2019-03-24]. Dostupné z: https://flaviocopes.com/react-server-side-rendering/

[16] Spring Data JPA - Reference Documentation. *Spring* [online]. San Francisco: Pivotal Software, 2019 [cit. 2019-04-09]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/